Advanced Engineering Technology and Application
An International Journal

Extension in The Case of Arrays in Daikon like Tools

M. H. Fouladgar*[a], B. Minaei-Bidgoli[a], H. Parvin[b], H. Alinejad-Rokny [c, d]

[a] *School of Computer Engineering, Iran University of Science and Technology (IUST), Tehran, Iran*
[b] *Department of Computer Engineering, Nourabad Branch, Islamic Azad University, Nourabad, Iran*
[c] *Complex Systems in Biology Group, Centre for Vascular Research, Faculty of Medicine, The University of New South Wales, Sydney, NSW, Australia*
[d] *School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW, Australia*

Email: Parvin@iust.ac.ir and H.Alinejad@ieee.org

Received: Sep. 10, 2012; Revised Dec. 14, 2012; Accepted Feb. 16, 2013
Published online: Mar. 1, 2013

Abstract: Software engineering comprises some processes such as designing, implementing and modifying of code. These processes are done to generate software fast and have a high quality, efficient and maintainable software. In order to perform these processes, invariants can useful and help programmers and testers. Arrays and pointers are frequent data types and are used in program code repeatedly. Because of this conventional use, these two data types can be the reason of fault in some program codes. First and last elements of arrays can confront to fault because of carelessness in using index in loops. Also arrays with the same type mostly have some relations which can be probably faulty. Therefore invariants which can report array and pointer properties are functional. This paper presented some constructive extension to Daikon like tools so that can produce more relevant invariants in the case of array.

Keywords: Dynamic invariant detection; software testing; array property; array's first & last elements; mutual element between arrays.

1. Introduction

Invariant are program valuable properties and relations which are true in all executions. For example in a sort function such as bobble 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 in [1]. Since invariants repeat the properties and relations of program variables, invariants can express the behavior of a program [4], 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*. Since specification and documentation are essentials in software engineering, Invariants can be used in all processes of software engineering from design to maintenance [2]. There are two different approaches to detent invariants, *static* and *dynamic*. This paper focuses on dynamic invariant detection.

Dynamic invariant detection is first time introduced by *Daikon* [3] - a full-featured and robust implementation of dynamic invariant detection. Daikon is the most prospering tool for detecting dynamic invariant and until now, comparing with other dynamic invariant detection methods [3]. Most of other tools and method are inspired by Daikon. Though Daikon is potent, one of the greatest problems of this tool is being time-consuming.

One of the most time consuming parts of software engineering is testing because regarding to different inputs, different paths in execution happen. However tester tries to test all different paths by different inputs, unchecked paths can be faulty. In this situation, because of their structure, *arrays* and pointers are more probable to be faulty. By means of invariants, programmer

or tester can recognize the behavior of program. Invariants detection tools report the properties and relations among variables. These properties and relations can be use in code testing after each up-date. Therefore if any improvement is achieved for the reporting some relevant invariant about arrays can help tester to find out program fault.

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.

2 Related Work

In this section, we discuss some implementations of dynamic invariant detection. We mention some implementations which are more relevant to our job but it is worth to mention there are many valuable efforts in this topic.

Dynamic invariant detection is first time introduced by *Daikon* [3] - a full-featured and robust implementation of dynamic invariant detection. Daikon is the most prospering tool for detecting dynamic invariant and until now, comparing with other dynamic invariant detection methods [3]. Most of other tools and method are inspired by Daikon. Though Daikon is potent, one of the greatest problems of this tool is being time-consuming.

DySy [8] is a dynamic inference tool which uses dynamic symbolic execution to expand the quality of inferred invariants. In the other words, besides executing test cases, DySy contemporarily perform a symbolic execution. These symbolic executions cause to produce program path condition. Then DySy combines the path conditions and build the final result. The result includes invariants which are expressed according to the program path condition.

Agitator [9] is a commercial testing tool and is inspired by Daikon. Software agitation was introduced by Agitar. Software agitation joins the results of research in test-input generation and dynamic invariant detection. The results are called observations. Code developer checks these observations to find out if there is any fault in the code. If there is any fault programmer or tester remove it and so on. Agitar won the Wall street Journal's 2005 Software Technology Innovation Award.

The DIDUCE is a dynamic invariant inference tool which extracts not only program invariants but also helps programmer to detecting errors and to determine the root causes [10]. Besides detecting dynamic invariant, DIDUCE checks 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 Paper Contributions

One of the most time consuming parts of software engineering is testing because regarding to different inputs, different paths in execution happen. However tester tries to test all different paths by different inputs, unchecked paths can be faulty. In this situation, because of their structure, *arrays* and pointers are more probable to be faulty. By means of invariants, programmer or tester can recognize the behavior of program. Invariants detection tools report the properties and relations among variables. These properties and relations can be use in code testing after each up-date. Therefore if any improvement is achieved for the reporting some relevant invariant about arrays can help tester to find out program fault.

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:

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

4 Clarifying of Contributions

To simplify and clearing up the contributions we talk over before, in this section, we illustrate our idea by some pieces of program code and their post-condition invariant. We state the *Exit* program point invariants which represent post-condition properties for a program point because post-condition properties can determine both the pre-condition and post-condition values of variables.

Now we introduce first paper contribution. In order to determine probable faults in arrays we employ some of first and last elements of array to invariant detection. 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.

To clarify our contribution considers Figure. 1. This figure shows a faulty version of bubbleSort(). It accepts 2 values as input. One of which is the array and another is the length of the array. The output is the sorted array. This version of bubbleSort has a fault. The index *j* starts at 1 instead of 0 so the first element of array is not considered in the sorting. By using of the first and last elements of the array in invariants detection, some useful invariants are produced which help us 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];
        nembers[j]=numbers[j+1];
        numbers[j+1]=temp;
      }
  return numbers;
}
```

Figure 1: Program A: Inattention in using index

By employing the first and the last elements of array in invariant detection, related invariants in the *Exit* point of the bubbleSort() of Figure.1 is shown 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 14 expresses that the first element of the input array always equals to the first element of the return value. Lines 15 to 20 express 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.

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

Figure 2: Related invariants to the code of Fig 1

Another contribute we discuss in this paper is the number of mutual elements of same type arrays for each program point. It helps programmer to test the code in situations that an array is generated as a result of performing some activities on another array. To illustrate the idea you may consider function in Figure.3. This function accepts 4 parameters as inputs. The first parameter is a sorted array and others are respectively array length, the value of element which must be replaced, and the new value, respectively. This function replaces *m*'s value with *n*'s value as a new value.

```
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"

Exit point invariants of replace() is shown in the Figure.4. In this figure, invariants in lines 6 and 7 express the number of mutual elements between *d* (the first parameter of the function) and the return value. The number of mutual elements between *d* and *return* value must be equals to the number of mutual elements between *orig(d)* and the *return* value (line 6). Also, the number of mutual elements between *d* and the *return* value 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) < d[-1]
11 orig(l) < 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 4

5 Actual Example and Justification

Now, we plan to illustrate our ideas in some actual examples. We intend to know how our idea can practically help programmer to detect faults and their line of code. To do this, we study some rather small and simple subprograms which are caused "gold standard" invariants [9]. 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. The presented code does not assume the use of any specific programming language.

5.1 Try-Catch and Effectiveness of the Ideas

Try-Catch statements, which prevent program from facing to a halt, can be a point of fault. Function AVG(), which is shown in Figure.5, contains a *Try-Catch* statement. 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.

```
 1 int* AVG(int* a,int l,long* sum,int n)
 2 {
 3   int i,j,*numbers,temp;
 4   numbers=malloc(sizeof(int[l]));
 5   numbers[]<-0;
 6   Try
 7   {
 8     *sum=0;
 9     for(i=0;i<l;i++)
10     {
11       *sum=a[i]+*sum;
12       if(n!=0)
13       {
14           temp=n/5;
15           a[i]/=temp;
16       }
17     }
18     numbers=Sort(a,l);
19   }
20   Catch(e)
21   {
22     *sum=0;
23   }
24   return numbers;
25 }
```

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

In Figure.6, the related invariants in the *Exit* program point of the function are shown. As before, invariants are in the form of Daikon output but here we add also our proposed part. AVG() does sort the input array and return a sorted array as we see in the line 9 of Figure. 6. This invariant merely express that the program seems 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. Line 17 express that mutual elements between a[] and the return values can be zero. In line 34 we observe that the mutual elements between a[] and the return values can be less than l whereas it is expected to be equal to l. consequently, we find out that the program does not work properly and in some cases we do not have sorted elements of a[] in the return array.

5.2 A Comparison with Original Daikon

Now in this subsection, we compare our result with result of original Daikon. We plan to do comparison in the function Figure. 1. In Figure.1 we presented a faulty version of "bubble sort". In Figure. 2 we showed our the related invariants generated by a modified version of Daikon (a version of daikon which we add our idea to it). Now in Figure.7, the original Daikon invariants of this subprogram are presented.

```
 1 a[] > return[] (lexically)
 2 a[] >= return[] (lexically)
 3 a[] == orig(a[])
 4 sum > return[] (lexically)
 5 sum >= return[] (lexically)
 6 orig(l) == size(return[])
 7 return != null
 8 return[] elements >= 0
 9 return[] sorted by <=
10 return[0] >= 0
11 return[1] >= 0
12 return[2] >= 0
13 return[3] >= 0
14 return[-2] >= 0
15 return[-3] >= 0
16 return[-4] >= 0
17 Mutual(a[],return[]) >= 0
18 a[] elements >= return[0]
19 a[] elements >= orig(n)
20 sum > Mutual(a[],return[])
21 sum > orig(l)
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[-2] <= return[-1]
34 Mutual(a[],return[]) <= orig(l)
35 orig(l) < a[-1]
```

Figure 6: Related invariants for the code of Figure 5

As we see in Figure.7, original Daikon invariant do not help us to determine the fault. Despite the reality, line 6 and 9 express that the program works properly and returns the sorted array.

```
 1 ..bubbleSort():::EXIT
 2 digits[] >= return[] (lexically)
 3 digits[] == orig(digits[])
 4 orig(length) == size(return[])
 5 return != null
 6 digits[] elements <= return[orig(length)-1]
 7 return[orig(length)-1] in digits[]
 8 digits[orig(length)-1] in return[]
 9 return[] elements <= return[orig(length)-1]
10 orig(length) < digits[orig(length)-1]
11 orig(length) < return[orig(length)-1]
12 digits[orig(length)-1] <= return[orig(length)-1]
```

Figure 7: Related invariants to the bubblesort code of Figure 1 using original Daikon

6 Evaluation

In this section, we plan to evaluate our proposed idea. In order to reach this goal, we intend to come up with two kind of comparison between modified Daikon and Original one. At first we evaluate the running time and time order of both version of Daikon. Then we measure the quality of produced invariants by using of *relevance* [5]-[6].

The running times of the proposed modified Daikon and the original one in terms of millisecond is shown in the Figure.8. As seen, the time-order of both modified and original versions of Daikon are linear. In other words, by adding our idea to the original Daikon the time

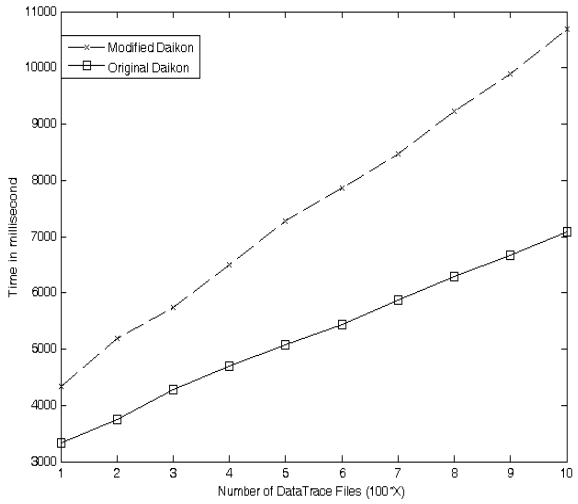

Figure 8: Time order of code of Figure 1 using different numbers of Data-trace files

order remains linear. However as there are more variable to check, modified Daikon has the higher slope of time order.

From another perspective, the relevance of the modified Daikon over some typical programs is summarized in Table.1. We study some rather simple and small pieces of program. Our reason is 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.

Now, consider Table.1. Rows are some different rather simple programs which we discussed some them in previous sections. Columns are representative of quality of invariants. As expected, 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.

Table 1: Relevance of modified Daikon in some case studies

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

7 Conclusions

In this paper, we discussed invariant as a significant entity in software engineering in recent years. Invariant detection tools report properties of program variables and relations between them. Since useful properties lead to more relevant invariants, we try to introduce two new properties of arrays which can cause new kinds of invariants. We focused on arrays because arrays are very conventional data structures which are used in all programs. As most of faults happen 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. After introducing these two ideas, we added them to Daikon. Daikon is a robust dynamic invariant detection tool. Then we evaluate our idea by comparing modified Daikon with original one. As mentioned, the time order does not change and it remains linear but with higher slope. Then we showed that more than 76% of inferred invariants are proper and relevant.

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. Assigning meanings to programs. *In Symposium on Applied Mathematics*; 1967; 19-32.

[2] Ernst M. D, Cockrell J, Griswold W. G, Notkin D. Dynamically discovering likely program invariants to support program evolution, *IEEE TSE;* 2007; 27(2): 99-123.

[3] Weiß B. Inferring invariants by static analysis in KeY. *Diplomarbeit, University of Karlsruhe, March*; 2007.

[4] Csallner C. DySy: Dynamic symbolic execution for invariant inference. *In Proc. of ICSE*; 2008.

[5] Ernst Michael D, Czeisler Adam, Griswold William G, Notkin David. Quickly detecting relevant program invariants. *In ICSE, Limerick, Ireland*; 2000; 7-9.

[6] Ernst M. D, Griswold W. G, Kataoka Y, Notkin D. Dynamically Discovering Program Invariants Involving Collections. *Technical Report, University of Washington*; 2000.