Tool Evaluation : DAIKON

Amol Prakash, Mausam

12th March 2002

1 Introduction

Daikon is a tool for dynamically detecting likely invariants in Java and C/C++ programs. It is built at UW by Michael Ernst. The Daikon system examines the values computed during the target program's execution, looking for patterns and relationships among those values.Properties that hold over the traces and also satisfy other tests, such as being statistically justified, not being over unrelated variables, and not being implied by other reported invariants, are reported as likely invariants. Like other dynamic techniques such as testing, the quality of the output depends in part on the comprehensiveness of the test suite. If the test suite is inadequate, then the output indicates how, permitting its improvement. Dynamic analysis complements static techniques, which can be made sound but for which certain program constructs remain beyond the state of the art. We have evaluated it for a C/C++ frontend on linux platform.

2 Experiments

We did experiments in the following three domains:

1. We implemented different data structures and algorithms associated with them. In particular we implemented sorting, binary search, stack, queue, list, linked list, heap, array addition, matrix addition, factorial etc.
2. We took the stack implementation of 30 students in CSE143 and attempted to evaluate the correlation between the quality of the code with the quality of the invariants produced by Daikon.

3. We also evaluated the extent of Daikon by using it to various C++ features like structs, pointers, passing by references, simple arrays, 2D arrays, recursion.

3 Problems encountered in Instrumentation

The front-end for C++ programs (dfec) is extremely buggy. Instrumented code usually has compilation errors. We are enlisting the various areas where instrumentation needs improvement:

- If we use main(int argc, char *argv[]), a small type mismatch is produced which can be quickly rectified in the instrumented code.
- Instrumented code does not dump the contents of the arrays if the array is global. To make it dump we passed them as parameters in the functions.
- Instrumented code does not dump the contents of the arrays if the array is declared within a class. The only thing we could find to make it work was to remove all the classes.
- Instrumented code does not dump the value of the pointers, hence no invariants could be found that used them.
- Instrumented code does not traverse the pointers for other memory locations. Hence linked lists didn't work.
- Instrumented code crashes if we compare two pointers. We had to manually change "x<=y" to "(&(*x))<=(&(*y))" to make it work!
- Instrumented code is incapable of handling the passing by reference and we couldn't rectify that.
- If we delete a pointer, there are compilation errors in the code produced. In this case, we removed all deletes and ran it again.
- Instrumented code crashes on a *new* for a pointer to pointer like int**. We could not rectify this.

4 Observations on invariants produced

4.1 Stack (implemented as array)

We implemented the stack ADT with all its usual functions in the array. This was the case which generated the best results for Daikon.

- All the expected invariants for push and pop were found accurately including the conditions for the failure of these functions (like stack full, or empty), relationship in the contents of the stacks before and after the functions, the return value of the pop function (which is the element popped), the change in value of top variable etc.

- We had used dynamic memory for array. But invariants like "*::Stack[::maxSize..] elements ==0*" showed that Daikon made an attempt to access the memory which is not a legitimate part of this program (as Stack is an array 0..maxSize-1).
- Similarly two confusing invariants were outputted, which also seems to be a result of the same memory violations:
  - "*::maxSize <= size(::Stack[])*"
  - "*::maxSize >= size(::Stack[])-1*"
- Consider the following function for determining whether Stack is empty or not:

```
bool isEmpty()
{ return top<0 ; }
```

Following are the relevant invariants outputted on its exit:
::Stack == orig(::Stack)
::Stack[] == orig(::Stack[])
::top == orig(::top)
::maxSize == orig(::maxSize)
::Stack != null
::top >= -1
return one of { 0, 1 }
::top < ::maxSize
::top <= size(::Stack[])-1
::maxSize <= size(::Stack[])
::maxSize >= size(::Stack[])-1

Hence one can see that the actual reasons as to why the function returns 1 or 0 is not at all captured. We could find no reasons as to why this happens. Similar things happen in the isFull function as well.

4.2 Selection Sort

- The sort function has all the necessary invariants including the fact that the array is sorted at the end.
- The checksort function which checks whether the array is sorted works fine when return value is 1. Of course "not sorted" cannot be made a primitive and so it only captures a single conditional:
  - *"(return == 1) ==> (A[] sorted by <=)"*
- The main function does not include the invariant of the array being sorted which shows that the instrumentation does not dump the global array at the exit of the main.

- We implemented a step by step selection sort. Consider the following selection phase which selects the ith smallest element.

```
void selsort(int *A, int i, int size)
{
   int smallest=Inf;
   int base=0;
   for (int j=i; j<size; j++)
   { if (smallest>A[j]) { smallest =A[j]; base=j;} }
   A[base]=A[i];
   A[i]=smallest;
}
```

One can make the following observations regarding the invariants generated for this function:

- The invariant does not say that A[0..i+1] is sorted. It only mentions that A[0..i] is sorted which was the precondition as well.
- The invariants nowhere reflect the fact that A[i]=min(orig(A[i..size-1])).
- The fact that A[0..i] remain unchanged in mentioned as : "*A[0..i-1] is a subsequence of A[0..i]*" which is a weaker statement.
- There was no mention of the local variables (base and smallest) in the invariants which is why the invariants could not express the fact that the original A[i] is swapped with A[base].
- There were some instances of redundancy for example
  - "*A[0..i] sorted by <=*"
  - "*A[0..i-1] sorted by <=*"

4.3 List (implemented as array)

We implemented a list with regular insert, delete, search functions. This list was implemented as a list with no-duplicates allowed. Insert always appended to the list. Delete always transferred the last element to the element which was getting deleted. The following observations could be made:

- Daikon got the idea of insert quite i.e. it realised that list has no duplicates, that insert returns 0 if data is already present in the list, or if the list if full and when it returns 1, the data is appended at the last available position. It, of course didn't get the idea that it returns 1, when data is not present in the list. Sadly, it couldn't combine to output the general invariant: "(return==0)==> (size=Max)OR(data in Array[0..size-1])".
- Similarly for search, it realised when the search returns 1 but could not get when it returns 0.

- It couldnot understand the new state of the array after the successful delete.

4.4 Array Addition

We tested Daikon for array addition. Small arrays were taken as test suit cases. Array contents were not dumped if the whole array was passed as a parameter. We then instead passed the pointer, which worked. We observed the following good invariants :

- The array size is same for all the three arrays.
- At the entry point of the funtion for adding elements at index i :
  "$a[0..i-1] \leq c[0..i-1]$"
  "$b[0..i-1] \leq c[0..i-1]$"
  where c is the array which stores the sum.
- At the exit point of the funtion for adding elements at index i, the invariant that says the the contents of array c remain unchanged except for index i.
- At the exit point of the funtion for adding elements at index i :
  "$a[0..i] \leq c[0..i]$"
  "$b[0..i] \leq c[0..i]$"
  where c is the array which stores the sum.

The following expected invariants were missed :

- At the exit point of the main function :
  "$a[] + b[] = c[]$"
- At the entry point of the funtion for adding elements at index i :
  "$i \leq size(A) - 1$"
- At the entry point of the funtion for adding elements at index i :
  "$a[0..i-1] + b[0..i-1] = c[0..i-1]$"

4.5 Matrix Addition

We tested Daikon for matrix addition. Small matrices were taken as test suit cases. We used a two-dimensional array implementation for matrices. The array contents were not dumped if the array was passed as a parameter. We then instead passed the pointer, which worked. We could not use a pointer to pointer to make dynamic 2-d arrays. Daikon crashed if we tried doing so. In this case, Daikon gave all the results treating it as a 1-d array.
We observed the following unexpected results :

- There was no mention at any point of the invariant (or something similar to it) :
  "$a[] + b[] = c[]$"

- We expected the following invariant at the entry point of the funtion for adding elements at index $[i, j]$:
  "$i \leq size(A)$ -1"
  "$i \leq size(B)$ -1"
  "$i \leq size(C)$ -1"
  "$j \leq size(A)$ -1"
  "$j \leq size(B)$ -1"
  "$j \leq size(C)$ -1"
  whereas the above were inferred at the exit point.

- At the exit point of the funtion for adding elements at index $[i, j]$, one invariant inferred all elements of a and b to be less than c. We expected this to be true only till index $i * columns + j$ and not for whole array.

4.6 Passing by reference

We tried writing a code to swap two numbers by passing by reference. The instrumented code did not compile. This seems to be a bug with the instrumenter dfec.

4.7 Binary Search

This code basically had two functions : one to sort the array and the other two do a binary search for the sorted array. We observed the following behaviour :

- Daikon could infer that the array was sorted by $\leq$ operator. But it could do this only when the array was passed as a pointer to the function. If the array was global, even though its contents were being dumped, still this invariant was not being generated.

- Daikon could not infer for the search function as to when the return value would be true and when would it be false. We tried to give a larger test suite with a variety of cases, but still this invariant could not be generated.

4.8 Circular Data Structure

We implemented circular queues using arrays to check for a circular data structure. We were expecting results similar to stack, but some of the results did amaze us :

- Trivial invariants were inferred for the tail and head in the case of push and pop respectively. We expected some kind of a circular behaviour to come out in the invariants, but the invariants just said that the head and tail changed after the operation.

- The two return branches for functions IsEmpty and IsFull could not be talked about differently through invariants i.e. the cases when true is returned and the the cases when false is returned was not talked about.
- Rest all invariants including the ones that dealt with the contents of the queue were found.

4.9 Queue (implemented as Linked List) and Linked List

We implemented the queue as a linked list but it Daikon, it seems, can't really handle linked lists, as we could see that the trace file generated doesnot have any mention of the contents of the list. Hence one did not find any invariants which would have otherwise found in a queue data structure.

Moreover, the top and end (which are queue*) are also not present in the invariants which shows that Daikon doesn't even trace the values of independent pointers. So the usual invariants like "empty=1 when top=NULL" or "after add, end always changes" were also not found.

We had run the delete function only when the queue is non-empty. The Daikon showed this as the precondition of delete: "*::size>=1*". Hence if a test-case is insufficient to handle some cases Daikon reports appropriately.

Similarly for linked list, no invariants could be found at all. It was quite certainly a blank file of invariants.

4.10 Heap (implemented as array)

The heaps were implemented in the array just to observe the extent of the functionality of the tool. We used the CLR implementation of the heapify and build_heap.

- The heap property at the of the build_heap could not be captured as expected.
- There were 80 invariants generated for heapify. Some of these mentioned the basic issues like some array not changing etc. Most others were just too unrelated to the our original expectations and were not really useful. Only other somewhat meaningful ones are being enlisted below: (Note that heapify creates a heap rooted at i):
  - "*A[i+1..] <= orig(A[i+1..]) (elementwise)*"
  - "*A[0..i]>=orig(A[0..i]) (elementwise)*". [Basically only A[i] has changed of these.]
  - "*A[0..i-1] is a subsequence of orig(A[0..i])*" [which is its way of saying that A[0..i-1] haven't changed!]

This shows that the partial heapify property is not really understood.

4.11 Recursive functions

We tested on three recursive functions: factorial, power and multiK (multiply a number n by k by adding n recursively k times). Since factorial and power are not primitive operators defined in Daikon, those invariants were obviously not generated. Now consider the multiK function:

```
int MultiK (int n,int k)
{
  if (k==0) return 0;
  else return n+ MultiK(n,k-1);
}
```

Even for this the usual invariant of "return=nk" was not found. This should have been found because Daikon checks for all possible relationships among three variables.

Note however that for all the functions the base cases were found separately. For example, for fact, the final exit looked like this:

std.fact(int;):::EXIT
n == orig(n)
(n == 0) ==> (return == 1)
n >= 0
return >= 1
n<=return

5 Testing on CSE143(C++) programs

Daikon was tested on a couple of CSE143 (C++) submissions. The aim was to see if Daikon can be used as a tool while debugging and grading.

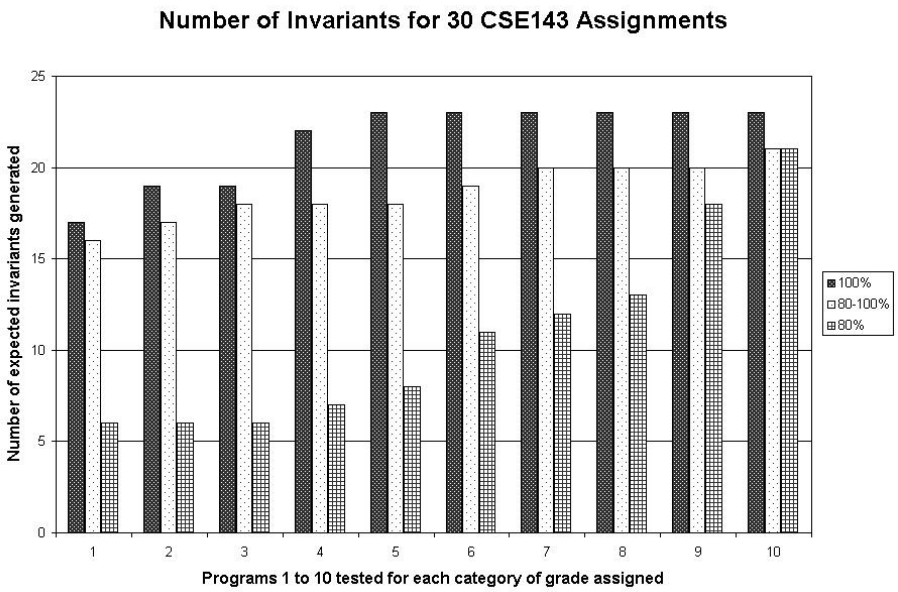
The assignment was to implement a stack using dynamic arrays and then to build a Craps game over that. The stack would be used to keep track of players. The concepts of Object Oriented Programming were to be exploited, so the students were to code five classes : Stack, Player, Crapstable (implementing rules of the game), Dice and House (the main class)

The test suite was chosen as follows :

10 submissions were taken each for grade less than 80%, grade between 80-100% and perfect grade. We observed the following things :

- Since classes were used heavily, Daikon could not get the trace of the data members (non-basic data types) of the classes eg. the contents of the array used for stack was not dumped. So, many significant invariants could not be found. Other than these, most of the assignments with perfect grades had all the invariants that one would have expected. These invariants basically involved simple relations between data members of the class.

- For many assignments with non-perfect grades, there were no traces for many implemented functions. This was due to programming bugs. Though very trivial, but this can be used as the first step in debugging the assignment i.e. ensuring that all the functions are being called.
- Again due to programming bugs, many a times the invariant found at the function exit was unexpected. For some funtions, one branch was never taken (due to bugs) because of which the invariant that was inferred was not the correct one. This could be used as an important tool for debugging. If some unexpected invariant comes up, it shows that the code has bugs.
- To help with grading, it can be checked as to how many of the expected invariants were generated for an assignment. Though not perfect, this could help in evaluating the program. The total number of expected invariants were 23. As can be seen from the figure, where each bar represents a program, in general the grading guidelines are good. The program which generated more invariants got more grades in general, but we do see some exceptions. There is a program which generated just 17 invariants out of 23, but still it got a perfect score. Another program which generated 21 invariants out of of 23 managed a score less than 80%.



6 Overall qualitative assessment of the invariants.

- Redundancy is not completely eliminated.

- There are times when Daikon does not seem to be consistent. For example, in the case of array addition, some preconditions like "i<=size" were missed but the same were present in the case of matrix addition.
- Daikon is unable to handle circular data structures (which have a mod size operation).
- Many of the primitive operators are justifiably unidirectional (ex in, sorted by, etc). So whenever we need not-sorted, or not-in, then Daikon fails.
- Daikon is unable to handle any invariants which require the values of the pointers or requires to traverse different pointers. It is more probably an instrumentation error as the values are not really dumped in the trace file.
- Daikon, in its current version is totally unable to determine meaningful invariants of the trees, in arrays.
- It seems only one "i-1" is allowed in Daikon invariants. So all invariants of the form "Array[0..i-1]==orig(Array[0..i-1])" are outputted in its weaker form "Array[0..i-1] is a subsequence of orig(A[0..i])".
- Daikon fails to recognise elementwise relationships between three arrays, like "A[]+B[]=C[]" was missed in case of array addition.
- Accessing illegitimate memory is observed when dynamic memory is used.
- In many cases, a very large number of invariants were observed out of which only few were meaningful to the programmer, though it seems a hard problem to determine what might be meaningful and what not.
- Invariants using the local variables were not outputted.

7 Overall comments on the usage of the tool

- Scalability: Daikon invariant generation does not seem to be optimised for large memory search. So, running it on large test suites involving arrays does not work. It should rather be used on smaller test suites.
- Instrumentation: Though the invariant generation process seems more or less bug-free, front end for C++ programs has a lot of bugs, and it is crucial to correct the instrumentation of code, to be able to use it practically.
- Debugging: Searching for expected invariants and presence of unexpected ones can help in the debugging. Also, it can help identify portions of code which are never executed. This may further help in the knowledge of any bias in the suits and so one can improve them to generate better invariants and do better debugging.

- Extent: In its present form, Daikon seems useful only for quite simple programs. To actually make it useful for a real-time programming scenario, it needs to be extended to handle more complicated programs.
- Grading: The number of invariants generated can help give a qualitative analysis of the code functionality. Though not very accurate, but this can surely help while making grading guidelines, and also to look for specific portions of code while grading to find the errors.

8 Ideas for improvement

- If an unusual function appears in some portions of the function (like sorted by, in, etc) then one may check it for always not-holding in other portions of the function.
- Correcting the pointer functionality. In its current version Daikon is incapable of handling them appropriately.
- Adding the functionality of various usual implementations of trees, in arrays.
- Finding relationships between variables in Daikon is quite adhoc. Rather, one may do the static checking of the program to determine a big possibility of relationships and then test them in the dynamic fashion. This might help in saving the overload of the outputted invariants as well as help in adding some other useful invariants. For example in our matrix addition case seeing the term Array[columns*i+j] in the code, it may suspect some possible invariants.