GENERATING HIGH CONFIDENCE CONTRACTS WITHOUT USER INPUT USING DAIKON AND ESC/JAVA2

by

BALAJI RAYAKOTA

B.Tech, JNTU, Hyderabad, 2009

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2013

Approved by:

Major Professor
Dr. Torben Amtoft

Abstract

Invariants are properties which are asserted to be true at certain program points. Invariants are of paramount importance when proving program correctness and program properties. Method, constructor, and class invariants can serve as contracts which specify program behavior and can lead to more accurate reuse of code; more accurate than comments because contracts are less error prone and they may be proved without testing. Dynamic invariant generation techniques run the program under inspection and observe the values that are computed at each program point and report a list of invariants that were observed to be possibly true. Static checkers observe program code and try to prove the correctness of annotated invariants by generating proofs for them. This project attempts to get strong invariants for a subset of classes in Java; there are two phases first we use Daikon, a tool that suggests invariants using dynamic invariant generation techniques, and next we get the invariants checked using ESC/Java2, which is a static checker for Java. In the first phase an 'Instrumenter' program inspects Java classes and generates code such that sufficient information is supplied to Daikon to generate strong invariants. All of this is achieved without any user input. The aim is to be able to understand the behavior of a program using already existing tools.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my major professor Dr. Torben Amtoft for his constant guidance and help throughout the duration of this project. I would also like to thank Dr. Gurdip Singh and Dr. John Hatcliff for accepting to be in my committee. I would like to thank Dr. David Schmidt for giving his valuable suggestions.

I would like to thank my brother, Bharath Rayakota, and my mother for their constant support and motivation. I would also like to thank my roommates, Janu and Sathya Chandran, for providing a fun filled environment and for their riveting discussions which helped me to finish this project.

Dedication

This work is dedicated to my father, Appa Rao Rayakota, who taught me many interesting lessons about life.

Chapter 1 - Introduction

A software contract describes the behavior of code. It specifies what is assumed before execution of code, called precondition, what will hold true after code finishes execution, called postcondition, and what does not change because of execution, called frame conditions. As contracts specify behavior without any ambiguity, unlike coding comments, they are used to prove properties about code. Software contracts are specified using invariants, which describes behavior that does not change at a particular program point. Even with many inherent advantages of specifying software contracts they are not regularly implemented in software industry, chiefly because it takes lot of effort to write strong and useful invariants.

Automatic generation and checking of invariants is a popular research topic. Two popular techniques, to solve these problems, are dynamic analysis, which runs the program under inspection and stores values of variables to infer relationships between them, and static analysis, which analyses the code and tries to generate proof to support claims. Daikon is a tool which suggests likely invariants using dynamic techniques and ESC/Java2 is a static checker for Java which tries to validate claims on programs by generating proofs for it.

Goal

The goal of this project is to try to infer program behavior without any user input. To achieve this goal Daikon is given the input of code run with random inputs, the invariants thus obtained are tested against ESC/Java2 to gain confidence about inferred invariants.

Overview

Chapter 2 looks into the details of invariants and describes some research on invariant generation, although the research on invariant generation is not directly related to the project at hand it serves the purpose of introducing the current state of automatic invariant generation. Chapter 3 describes, in short detail, the working of the tools Daikon and ESC/Java2. Chapter 4 presents the implementation details of this project. Chapter 5 explains some of the results obtained as a result of this project. Chapter 6 provides a brief conclusion to the report along with future work which can be done to continue the work described in this report.

Chapter 2 - Invariants

This chapter explains the importance of invariants and brushes over some of the research done in the field of automatic invariant generation.

Definition and uses

Invariants are properties at program points which are claimed to be true no matter which input is supplied to the program. In this respect invariants are any property at specified program point which does not change. Perhaps the invariants that are most popular, and most troublesome to describe, invariants are loop invariants, which are properties which are claimed to be true at loop entry and exit.

Uses of invariants

Invariants are useful in all stages of software development life cycle. Uses of invariants are described in [1], a short summary of the points mentioned are written here.

Documentation: Invariants describe some properties of the program at a specified point; hence it can be used to describe program behavior and can act as a means of communication between developer and user or between other developers. Because invariants are specific in their meaning it is less likely to get confused about intended meaning.

Avoiding bugs: Invariants can be used to describe the assumptions made for the program to function properly, preconditions, also they can be used to describe the intended behavior of the program, post conditions, upon which some other piece of code might be depended on. Hence, writing invariants can avoid the situation in which programmer makes changes to the code which does not protect the pre and post conditions.

Debugging: Once a bug is involuntarily introduced into the system then it is the pre and post conditions which help in tracking and solving the bug. As invariants preserve these properties they help in debugging too.

Testing: Invariants help in the process of testing in two ways, they help in development of a test suite and they help to validate the output of a test suite.

Verification: Invariants help static checker in proving certain program properties. As an example it is almost impossible to prove program correctness for loops without loop invariants.

Program evolution: As described earlier invariants describe the pre and post conditions of each routine. This documentation of behavior helps in adaptation and evolution of programs by avoiding unnecessary bugs being introduced.

Research on automatic generation of invariants

This section describes some of the research being conducted on automatic generation of invariants. The chief bottleneck in generation of invariants for the whole program is loops, as it is still a hurdle to generate strong and useful loop invariants. This section describes three experimental methods of generating loop invariants.

Predicate abstraction for software verification

This section briefly introduces the work described in [2]. This paper describes a method to reduce the problem of developing loop invariants to developing a set of relevant predicates for the loop. It does this by inferring loop invariants that are a boolean combinations of these predicates. Thus, infinite concrete state space is abstracted to a finite abstraction domain. The invariant for each loop is obtained by iterative approximation. By abstracting a set of reachable states at loop entry the first approximation is obtained. To calculate the next approximation current approximation is enlarged by including the states reached by executing the loop body from the set of reachable states in current approximation. The iteration terminates in a loop invariant since the abstract domain is finite.

Example:

```
for (int i = 0; i<a.length; i++)
        a[i] = null
```

Predicates: 0<= sc, sc < i, a[sc] != null

Invariant: (\forall int sc; 0<=sc && sc < i ==>a[sc] = null)

Simplifying loop invariant generation using splitter predicates

This section presents the work described in [3]. The basic assumption in this paper is that it is easier to infer loop invariants without conditionals than to find for loop with conditionals. To

solve this problem loops with a single conditional statement may under certain conditions split into two separate loops without any conditional using the concept of splitter predicate. The rest of the paper describes a method to infer splitter predicates.

Example:

```
x=0; y=50;
while(x<100)
{
        x=x+1;
        if(x>50)
                y=y+1;
}
```

is split into

```
x=0;y=50;
while(x<=49)
{
        x=x+1;
}
while(x<100 & x>49)
{
        x=x+1;
        y=y+1;
}
```

This does the same job as the loop above.

Finding loop invariants for programs over arrays using a theorem prover

This section presents work described in [4]. In this method the authors introduce the concept of an update predicate which describes in which iteration an array is updated and by what value. Given a loop over array and scalar variables this method tries to extract all possible information about scalar variables and loop counter using techniques of symbolic computation

such as recurrence solving and quantifier elimination. Next, update predicates for each array is arrived upon. In the final step saturation theorem prover is used to eliminate auxiliary symbols and obtain the loop invariants expressed as first order formula.

Example:

```
a=0;b=0;c=0;
while(a<=k){
        if(A[a]>=0)
 {
        B[b]=A[a];
        b++;
 }
        else
                {
                        C[c]=A[a];
                        c++;
                }
        a++;
}
```

Properties:
b>=0, c>=0, a=b+c

Update predicates:
$upd_B(i,p,v)$: B is updated at position p with value v
Similarly update predicate for array A is evaluated.

Invariants using update predicates:
$(i \in iter)(A[a^{(i)}] >=0 => B^{(i+1)}[b^{(i)}] = A[a^{(i)}] ,$
$b^{(i+1+} = b^{(i)}+1,$
$c^{(i+1)}= c^{(i)})$
Final invariant:
$(\forall x)(b > x \wedge x > 0 => (\exists y)(A[y] = B[x] \wedge A[y] > 0))$

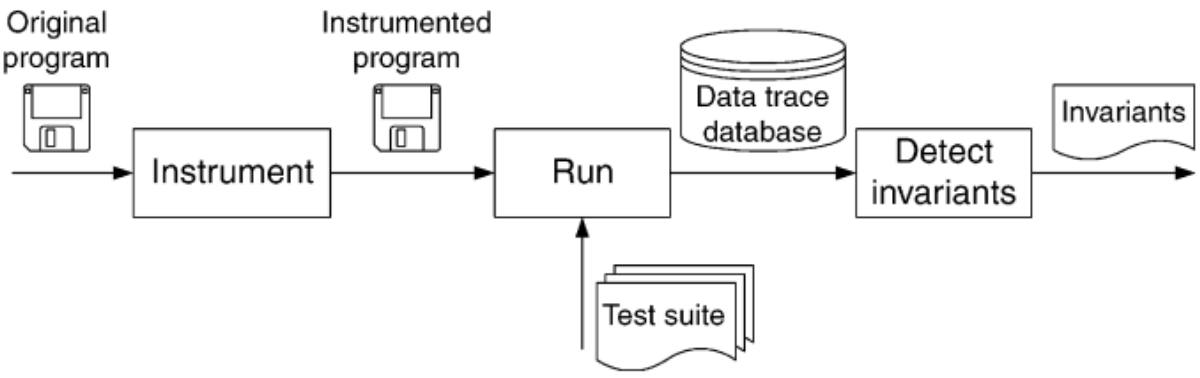
Chapter 3 - Daikon and ESC/Java2

This chapter briefly describes the workings of Daikon and ESC/Java2, which are the essential components of this project.

Daikon

Daikon is a tool which reports likely invariants for the input code at specific program points, like procedure entry and exit points, using dynamic invariant generation techniques. Daikon achieves this by first inserting commands at various program points to collect the values of variables, which are in scope at that point; these values are collected in a trace file and this whole process is called instrumenting of input program. Once necessary trace data has been collected Daikon's inference engine looks for patterns in the values of collected variables and reports the patters (invariants) which are observed to be true in all samples. To make sure that invariants which are observed to be true are not true by mere coincidence, confidence for each invariant is calculated and only those invariants which have high enough confidence are reported.

Figure 3.1 Architecture of Daikon (excerpt from [5])



Instrumenter

Daikon front end accepts program and instruments it by translating the source code to source code with additional commands, this process does not change the behavior of input program in any way. The instrumenter parses the source code to obtain the abstract syntax

tree(AST). From this AST the variables which are in scope at program points of interest are inferred and commands are inserted at these program points to output the value of these variables into a trace file. After this, the instrumentser unparses the AST to obtain a source code, which is then compiled in normal way. The modified code thus obtained is called as instrumented code.

As of the implementation of this project Daikon team has developed instrumenters for Java, C, C++, Perl, and Eiffel. Though for this project only the instrumenter for Java, Chicory, was used.

Test Cases and Invariant Detection

The instrumented program is now run with a test suite and the output, which is a trace file, is given as input to Daikon's inference engine, which is also called Daikon.

The job of Daikon's inference engine is to report presence of invariants in the program by checking values present in the supplied trace file. It has a list of predefined invariants to check against the values in the trace file. At the beginning of the process the inference engine assumes all invariants to be true and checks if any sample data invalidates any of the invariants in its list, if some invariant is invalidated then it is removed from the list of reported invariants. Thus, the basic algorithm being used is a test-and-check algorithm. Once all the samples in trace file are checked the remaining invariants are reported as likely invariants. The reported invariants are also calculated for confidence to make sure that they are not true by chance.

Types of invariants detected by Daikon

Types of invariants detected by Daikon. This section has been quoted from [5], page 102.

1. Invariants over any variable:
    a. Constant value: $x = a$ indicates the variable is a constant.
    b. Uninitialized: $x =$ uninit indicates the variable is never set.
    c. Small value set: $x \in \{a,b,c\}$ indicates the variable takes only a small number of different values.
2. Invariants over a single numeric variable:
    a. Range limits: $x \geq a$; $x \leq b$; and $a \leq x \leq b$ (printed as x in [a..b]) indicate the minimum and/or maximum value.
    b. Nonzero: $x \neq 0$ indicates the variable is never set to 0.
    c. Modulus: $x \equiv a \bmod b$ indicates that $x \bmod b \equiv a$ always holds.

d. Nonmodulus: $x \neq a \bmod b$ is reported only if $x \bmod b$ takes on every value beside a.

3. Invariants over two numeric variables:
    a. Linear relationship: $y = ax + b$.
    b. Ordering comparison.
    c. Invariants over x+y: any invariant from the list of invariants over a single numeric variable.
    d. Invariants over x-y: as for x+y
4. Invariants over a single sequence variable (arrays):
    a. Range: minimum and maximum sequence values, ordered lexicographically.
    b. Element ordering: whether the elements of each sequence are non-decreasing, non-increasing or equal.
    c. Invariants over all sequence elements (treated as a single large collection): for example, all elements of an array are at least 100.
5. Invariants over two sequence variables:
    a. Linear relationship: $y = ax + b$, elementwise.
    b. Comparison: lexicographic comparison of elements.
    c. Subsequence relationship.

For more details about working of Daikon refer [5], [6], [7].

ESC/Java2

ESC/Java2 is an extended static checker for Java which checks for programming errors which are not usually detected till runtime like null dereference, array bound errors, and type cast errors. The main intention behind developing a static checker is to push the step of debugging as early as possible in software development life cycle.

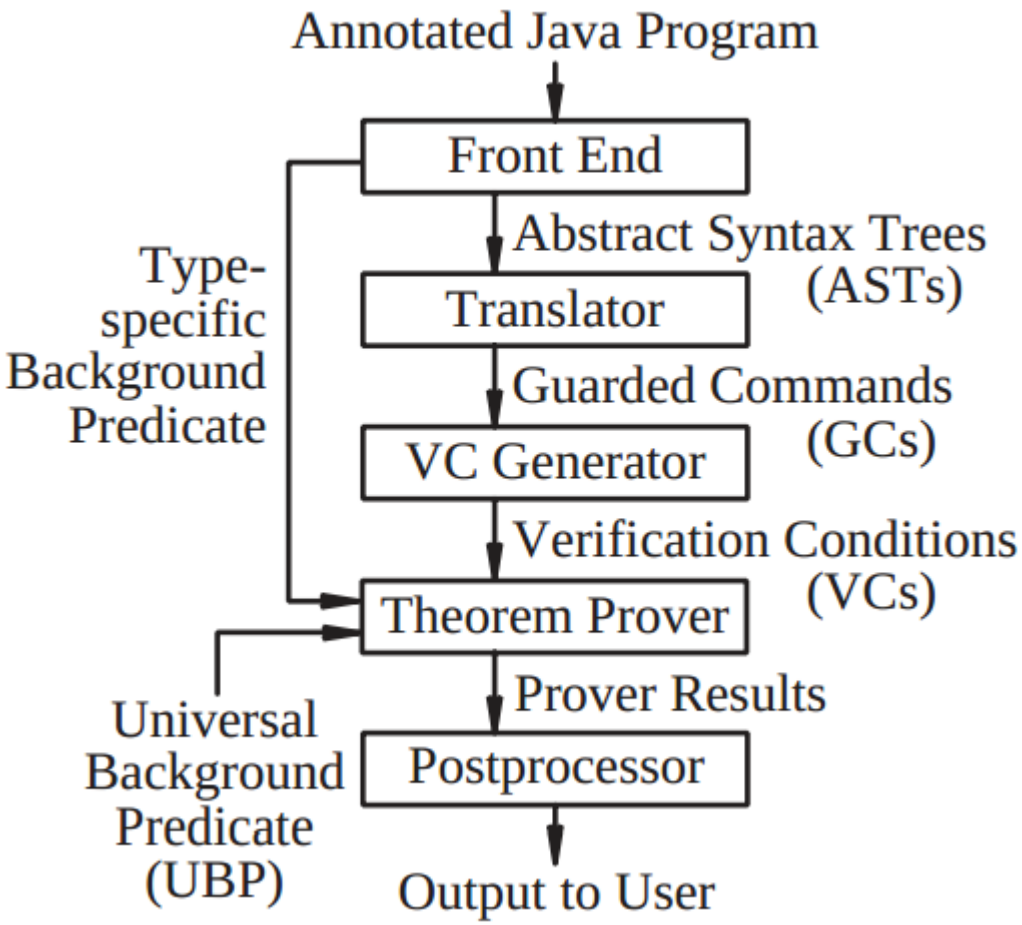
ESC/Java2 is weaker than theorem provers but more powerful than most type checkers. ESC/Java2 does not prove program correctness; rather it detects only certain types of errors. If the program is annotated with invariants then it tries to prove those annotations, failing to do so it issues warnings on those invariants.

In this project ESC/Java2 is used to check the invariants suggested by Daikon. ESC/Java2 is not sound nor complete which means that the warnings issued may be incorrect and that even invariants on which warning is not issued may be wrong. These short comings are not entirely shortcoming of ESC/Java rather they are tradeoffs made to make the tool faster and more usable.

Working of ESC/Java2

Following sections describe the functioning of ESC/Java which has been summarized from [8] and [9].

Figure 3.2 Architecture of ESC/Java2 (excerpt from [8])



Front End

The Front End parses the Java source code along with JML annotations to produce an abstract syntax tree (AST) and type-specific background predicate for each class whose routines are to be checked. The type-specific background predicates encodes information about the types used in Java code. The developed AST is fed as input to the translator.

Translator

The translator accepts AST and transforms the routines into a guarded command language, which is based on Dijkstra's guarded commands. A guarded command language has assertions before various commands and the execution is said to go wrong if at a specific program point these assertions are false. ESC/Java does modular checking which means that when it encounter procedure calls it depends on method specification rather than the procedure implementation.

VC Generator

This module generates the verification condition (VC) for each guarded command. A VC for a guarded command is a predicate that holds for precisely those program states from which no execution of the command can go wrong. What this means is that if by some means the VC can be proved to hold at that program point then the invariants are proved. VC is calculated using weakest precondition calculation.

Theorem Prover

ESC/Java's theorem prover (simplify) accepts universal background predicate (UBP), which are the semantics of Java encoded as predicates, and the type-specific background predicate and tries to prove if they together imply the VC generated for each guarded command. Failing to prove this, ESC/Java issues a warning.

Postprocessor

This module analyses the output of the theorem prover and issues warning for VCs which were failed to be proved, along with information about why it was not proved.

For more details about ESC/Java and ESC/Java2 refer [8] and [9].

Chapter 4 - Implementation

Problem with Daikon

As mentioned in previous chapter Daikon runs the code supplied to it and stores the values of variables accessible at various program points. After the program terminates Daikon inspects the values stored at those program points looking for patterns and relationships between various variables, Daikon has a list of patterns that it searches for in the stored values. Once the relationship between variables is established it reports the patterns which are always observed to be true as possible invariants. The output may not be universal invariants as the program run, or the test suite, may not reflect all the possible program outcomes.

A typical program uses its data structures, or abstract data types in Java, to accomplish its task or to achieve the desired results. One of the problems in using the entire program as an input to Daikon is that since the ADTs are not called numerous number of times the invariants suggested, tough may be correct for the present program, prove to be useless if one wants to reuse the ADTs and infer the intended behavior using the invariants.

Figure 4.1 Stack.java and UsesStack.java

```
public class Stack {

	public int[] S;
	public int top;

	public int isEmpty(){
		if(top == -1)
			return 1;
		else
			return 0;

	}

	public void push(int x){
		if(top < S.length-1)
			S[++top] = x;
		else
			System.out.println("Stack full!");
	}
```

Figure 4.2 Stack.java and UsesStack.java (continued)

```
public int pop(){
            if(top > -1)
                  return S[top--];
            else
                  return -1;
      }

      public Stack(){
            top = -1;
            S = new int[10];
      }



      public Stack(int size){
            S = new int[size];
            top = -1;
      }

}



public class UsesStack {
      public static void main(String[] args){
            Stack st = new Stack(5);

            st.push(3);
            st.push(4);
            System.out.println(st);
            System.out.println(st.pop());
      }
}
```

The above figure contains the code for an implementation of Stack data structure in Stack.java and one of the ways to use it in UsesStack.java.

Figure 4.3 Invariants of Stack.java on a bad test case

```
      /*@ invariant this != null; */
      /*@ invariant this.S != null; */
      /*@ invariant daikon.Quant.subsetOf(this.S,
                              new long[] { 0, 3, 4 }); */
      /*@ invariant this.top == -1 || this.top == 0
                                    || this.top == 1; */
      /*@ invariant daikon.Quant.size(this.S) == 5; */
      /*@ invariant daikon.Quant.eltsLTE(this.S,
                        daikon.Quant.size(this.S)-1); */
      /*@ invariant this.top<daikon.Quant.size(this.S)-1; */
      public int[] S;
      public int top;
```

Figure 4.4 Invariants of Stack.java on a bad test case (continued)

```
/*@
  @ public normal_behavior // Generated by Daikon
  @ requires true;
  @*/
  public int isEmpty(){
        if(top == -1)
              return 1;
        else
              return 0;

  }

  /*@
  @ public normal_behavior // Generated by Daikon
  @ requires daikon.Quant.getElement_int(this.S, x)
           == daikon.Quant.getElement_int(this.S, x-1);
  @ requires daikon.Quant.subsetOf(this.S,
                                   new long[] { 0, 3 });
  @ requires this.top == -1 || this.top == 0;
  @ requires x == 3 || x == 4;
  @ requires daikon.Quant.getElement_int(this.S, x) == 0;
  @ ensures this.S == \old(this.S);
  @ ensures \old(x) ==
          daikon.Quant.getElement_int(this.S, this.top);
  @ ensures daikon.Quant.size(this.S) ==
                        \old(daikon.Quant.size(this.S));
  @ ensures daikon.Quant.getElement_int(this.S, \old(x))
       == daikon.Quant.getElement_int(this.S, \old(x)-1);
  @ ensures daikon.Quant.getElement_int(this.S, \old(x))
        == \old(daikon.Quant.getElement_int(this.S, x));
  @ ensures daikon.Quant.getElement_int(this.S, \old(x))
       == \old(daikon.Quant.getElement_int(this.S, x-1));
  @ ensures this.top == 0 || this.top == 1;
  @ ensures daikon.Quant.getElement_int(this.S, \old(x))
                                                    == 0;
  @*/
  public void push(int x){
        if(top < S.length-1)
              S[++top] = x;
        else
              System.out.println("Stack full!");
  }

  /*@
  @ public normal_behavior // Generated by Daikon
  @ requires daikon.Quant.size(this.S)-1 ==
         daikon.Quant.getElement_int(this.S, this.top);
  @ requires this.top == 1;
  @ requires
    daikon.Quant.getElement_int(this.S, this.top-1) == 3;
  @ ensures this.S == \old(this.S);
  @ ensures
        daikon.Quant.pairwiseEqual(this.S, \old(this.S));
```

Figure 4.5 Invariants of Stack.java on a bad test case (continued)

```
        @ ensures this.top ==
                  daikon.Quant.getElement_int(this.S, \result);
        @ ensures this.top ==
                 daikon.Quant.getElement_int(this.S, \result-1);
        @ ensures \result == daikon.Quant.size(this.S)-1;
        @ ensures \result == \old(daikon.Quant.size(this.S))-1;
        @ ensures \result ==
           daikon.Quant.getElement_int(this.S, \old(this.top));
        @ ensures \result ==
           \old(daikon.Quant.getElement_int(this.S, this.top));
        @ ensures daikon.Quant.getElement_int(this.S, this.top)
                        == daikon.Quant.getElement_int(this.S,
                                              \old(this.top)-1);
        @ ensures daikon.Quant.getElement_int(this.S, this.top)
                   == \old(daikon.Quant.getElement_int(this.S,
                                                 this.top-1));
        @ ensures this.top == 0;
        @ ensures daikon.Quant.getElement_int(this.S, this.top)
                                                          == 3;
        @*/
        public int pop(){
              if(top > -1)
                    return S[top--];
              else
                    return -1;
        }

        /*@
        @ public normal_behavior // Generated by Daikon
        @ requires true;
        @*/
        public Stack(){
              top = -1;
              S = new int[10];
        }



        /*@
        @ public normal_behavior // Generated by Daikon
        @ requires size == 5;
        @ ensures \old(size) == daikon.Quant.size(this.S);
        @ ensures daikon.Quant.eltsEqual(this.S, 0);
        @ ensures this.top == -1;
        @ ensures daikon.Quant.eltsEqual(this.S,
        daikon.Quant.getElement_int(this.S, \old(size)-1));
        @*/
        public Stack(int size){
              S = new int[size];
              top = -1;
        }

}
```

If one glances at the invariants suggested by Daikon for push(x) method for the present program run it may be observed that it reports invariants like

```
@ requires daikon.Quant.subsetOf(this.S, new long[] { 0, 3 });
@ requires this.top == -1 || this.top == 0;
@ requires x == 3 || x == 4;
@ ensures this.top == 0 || this.top == 1;
@ ensures \old(x) == daikon.Quant.getElement_int(this.S, this.top);
```

which says that the method requires that elements of stack are subset of {0, 3}, it requires the value of top to be -1 or 0, the pushed value has to be either 3 or 4, in post-condition it is always the case that value of top is either 0 or -1, and the original value of x is equal to S[top]. Though these may be correct invariant for the current execution of Stack.java as used by UsesStack.java, it can clearly be seen that it does not reflect the behavior of Stack.java. Moreover if one wishes to reuse Stack.java and infers the program behavior from the invariants then there are high chances that the new program will malfunction.

Daikon suggests these invariants because for the amount of data it has been supplied with the patterns hold in all the cases. In above example run, it is always the case that push() is called with an integer parameter of either 3 or 4. To rectify this situation Daikon needs to be supplied with more distinct data about correct usage of Stack.java. This can be done by calling Daikon on a program which calls on various constructors and methods of Stack.java numerous times. This can be achieved by running the data structure on a good test suite. A good indication of a good test case is the coverage of code where the main aim is to call various methods and constructors with valid input and store the output in valid data types. As can be imagined, number of test cases used should not be limited as this will again result in triggering Daikon to report weak invariants.

So, to solve the problem of efficiently using Daikon to suggest invariants which reflects program behavior we need to run the ADTs we are interested in with valid test cases, keeping in mind that number of test cases used are enough to allow Daikon to weed out false invariants, and we need to check the invariants suggested by Daikon.

Problem with ESC/Java2

The only problem with ESC/Java2 is its dependence on annotation to prove program properties, which means that to use it to infer program properties one has to manually annotate the program.

Figure 4.6 Output of ESC/Java2 on Stack.java without any annotation

```
ESC/Java version ESCJava-2.0.5
   [0.076 s 42161912 bytes]



Stack ...
 Prover started:0.017 s 50342440 bytes
   [1.101 s 50943864 bytes]



Stack: isEmpty() ...
   [0.099 s 51698000 bytes]  passed



Stack: push(int) ...
------------------------------------------------------------
Stack.java:16: Warning: Possible null dereference (Null)
               if(top < S.length-1)
                         ^
------------------------------------------------------------
Stack.java:17: Warning: Possible negative array index
                                                  (IndexNegative)
                       S[++top] = x;
                        ^
Execution trace information:
   Executed then branch in "Stack.java", line 17, col 3.



------------------------------------------------------------
   [0.403 s 49702192 bytes]  failed



Stack: pop() ...
------------------------------------------------------------
Stack.java:24: Warning: Possible null dereference (Null)
                       return S[top--];
                              ^
Execution trace information:
   Executed then branch in "Stack.java", line 24, col 3.



------------------------------------------------------------
Stack.java:24: Warning: Array index possibly too large
                                                    (IndexTooBig)
                       return S[top--];
                              ^
Execution trace information:
```

Figure 4.7 Output of ESC/Java2 on Stack.java without any annotation (continued)

```
Executed then branch in "Stack.java", line 24, col 3.


------------------------------------------------------------
   [0.037 s 50464672 bytes]  failed


Stack: Stack() ...
   [0.017 s 50888312 bytes]  passed


Stack: Stack(int) ...
------------------------------------------------------------
Stack.java:35: Warning: Possible attempt to allocate array of
                                    negative length (NegSize)
               S = new int[size];
                           ^
------------------------------------------------------------
   [0.052 s 51481352 bytes]  failed
 [1.715 s 51481352 bytes total]
5 warnings
```

ESC/Java2 run on programs run on programs without any assertions just reports common programming errors. Without annotations ESC/Java2 does not check for any other properties.

To fully utilize ESC/Java2 we need to annotate code with assertions, which we think will hold, and use the tool to either prove or warn about it.

Proposed Solution

The goal of this project is to infer program behavior, or behavior of ADTs, without any user input. One way of achieving this is to use Daikon's to suggest likely invariants and to check the invariants by ESC/Java2.

The main obstacle in using Daikon is that any code that has to be tested should be run with proper test cases. To solve this problem, without any user input, a test case generator has to be built which can develop some test cases for the code to be tested. Next section will provide more details about the design of a random test case generator and what kind of code can be successfully tested by it.

After developing random test cases for the code we will be running it on Daikon, collecting the invariants it suggests. The last step will be to check the invariants so gathered by

ESC/Java2. By this process we will have a set of invariants which can be assured to be true with high confidence.

Note that in [10] the authors propose a similar solution, but not exactly the same solution as described in this report. The main problem addressed in this report is that of helping Daikon to suggest strong invariants and then statically checking the invariants, whereas in [10] the main problem addressed is that of developing a solution to statically check any invariants suggested by Daikon.

Random test case generator

This section will outline the design and implementation of random test case generator module which produces Java code to test Java ADTs.

Desired Output

Our main aim is to supply enough information to Daikon so that it can produce some useful invariants. Daikon collects information about variables at various program points like beginning and end of methods, beginning and end of constructors. So, in order to provide more data to Daikon we need to call methods and constructors numerous times, here numerous is a bit vague; according to tests conducted on Daikon the invariants suggested don't change after about 1000 calls of a particular method.

A typical Java class will specify the fields and behavior of its objects. Fields are usually declared to be private and behavior is specified in the form of constructors and methods. Thus, we want to call a class' public constructors and methods to supply data to Daikon. We are not interested in private constructors and methods because they are designed to be called only inside the class' code; also it is assumed that if we call public methods enough number of times they will in turn call private constructors and methods which should supply enough information to Daikon to suggest invariants for private methods and constructors.

In order to call methods and constructors we need to be aware of the parameter list and return type, which is not needed in case of constructors. In case of manual development of test suite various test cases are supplied by humans or by semi-automated method which require some human intervention. The main idea in this project is to develop test cases for code without any user input. To achieve this we will be calling methods with randomly formed parameters, which need to be of correct data type. To implement this idea we will be building code, called

'class block' which will instantiate objects of each class using each of the public constructor and with each such object every public method will be called; actually public methods will be called multiple times for each instantiation because they reflect the behavior of classes. Class block will be built for each class and these blocks will be executed multiple number of times.

The output of random test case generator will be code which can execute these class blocks.

Figure 4.8 Ouput of random test case generator

```
public class HasMain{
      public static int random(int Min, int Max){
      //code to return a random number in [Min, Max]
      }

      public static void main(String[] args){
            for(int loop = 0; loop < 200; loop++){
            {
                  //Class block for class 1
            }
            ....
            {
                  //Classblock for class n
            }
      }
}
```

Figure 4.9 Details of class block

```
      //constructor block
      <class name> v1 = null;
      //code to instantiate v1 with public constructors
      for(int cIter = 0; cIter < <no. of constructors>;
                                                cIter++){
            switch(cIter){
            case 0:
            {
               //instantiate variables which will populate
               //parameter list
               v1 = new <class name>(v2,.....);
               break;
            }
            ...
            case n:
            {
              //instantiate variables for parameter list
              v1 = new <class name>(v2,...);
              break;
            }
            //Method block
            for(int mIter = 0; mIter < <no. of methods>*50;
                                                mIter++){
                  int select = random(0, <no. of methods>);
```

Figure 4.10 Details of class block (continued)

```
            switch(select){
            case 0:
            {
              //instantiate variables to populate parameter list
              //create a variable of return type
              v<x> = v1.<method name>(v2,...);
              break;
            }
            ...
            case n:
           {
              //instantiate variables to populate parameter list
              //create a variable of return type
              v<x> = v1.<method name>(v2,...);
              break;
           }
       }
     }
   }
```

Next few sections will detail the process of developing this desired code.

Below is the code that will be the running example for random test case generator section.

Figure 4.11 A.java

```
public class A {
      public A(){
            System.out.println("Default constructor of A");
      }
      public A(int x, int[] y){
            System.out.println("Constructor of A, int
                                    parameter" + x);
      }

      public A(int[] x){
            System.out.println("Constructor of A, int[]
                                          paramter");
      }

      public void metA1(){
            System.out.println("A's met1, no parameters, no
                                          return type");
      }

      public int metA2(){
            System.out.println("A's met2, int return type");
            return 0;
      }
```

Figure 4.12 A.java (continued)

```
public int[] metA3(int a, int[] b){
            System.out.println("A's met3, int[] return type,
                                int and int[] parameters");
            return new int[8];
      }

      public void metA4(int[] a){
            System.out.println("A's met4, no return type,
                                    int[] parameter");
      }
}
```

Figure 4.13 B.java

```
//This is a test class
public class B {
      private B(int a){
      }

      public B(){
            System.out.println("B's defaulst constructor");
      }

      public B(A[] x, int[][] z){
            System.out.println("B's non default constructor,
                                     Type A parameter");
      }

      public void metB1(A x){
            System.out.println("B's method1, no return type,
                                     Type A parameter");
      }

      public A[] metB2(A[] x){
            System.out.println("B's method2,
                 Type A[] return type, Type A[] parameter");
            A[] y = null;
            return y;
      }

      private void met3(){
      }
}
```

Assumptions on input

For this project some restrictions on input classes have to be put to make the implementation possible in given time. There are few assumptions about input classes that were made which are as follows:

1. All the classes to be tested should be put in a single folder: this assumption was made to make the code easier to develop. If this experiment leads to successful results then in future versions provisions will be made to include multiple folders similar to Java's classpath specification.

2. Return type and parameter list types should be integers, integer arrays, one of the ADTs specified as input classes, or an array of above mentioned ADTs: we need complete information about the types used for return type and in parameter list. To make this possible this assumption has been made.

3. Classes should only have methods and constructors, in addition to fields: as of now only methods and constructor (both public and private) features are handled. Features like static code, inheritance, etc. will be handled in future versions.

Collecting data

One of the first things to be done with input classes is to collect relevant data about them which will help to produce desired code of class block, constructor block, and method block.

Information collected for each class:

1. Class name.
2. Constructor details.
3. Method details.
4. Preferred constructor to be used for instantiation.

Information collected for methods:

1. Method name.
2. Type details of return type.
3. Type details of parameter list.

Information collected for constructor:

1. Class name.
2. Type details of parameter lists.
3. Information content.

Information stored about types:

1. Type name.
2. Order of array.

To collect and store the required information about input classes java.reflect package was used. This package is used to get a ‘class’ object of the input classes and the required information can be queried using appropriate methods of that ‘class’ object.

Preferred constructor

When we have to instantiate an object of a particular class a problem arises when there are multiple constructors for that class and when it is not specified which constructor to use.

Possible approaches

1. Always use default constructor: This is the simplest approach which can be implemented and that might be the only advantage in using this approach. Disadvantage in adopting this approach is that the object created has no information in it and it might lead to Daikon suggesting weak invariants.

2. Select a non-default constructor: If non-default constructor(s) has/have been defined then select one of the non-default constructors to instantiate an object. Advantage of this approach is that we are not creating an empty object which will lead to better invariants being suggested. Disadvantage of this approach is the selection of a particular non-default constructor and the problem of recursive instantiation which has been explained below.

Problems faced:

1. Recursive instantiations: if a constructor requires as a parameter an object of the same class then it will lead to infinite loop of instantiation. One way to solve this problem is to create object of constant depth, i.e., when this situation arises keep track of how many objects have been created so far and in the end use the default constructor to create the last object.

2. Selection of constructor: The problem here is of deciding which non-default constructor to select if multiple constructors are present. To select one of the constructors there should be a mechanism to compare them and decide which suits better for our purpose.

Possible Solutions:

1. One criterion which can be used to compare the usefulness of a constructor is to be able to quantify the amount of information it is supplying to the object created. This cannot be done absolutely but a rough ordering between available constructors can be created by looking at the types of parameters they accept, this can be done assuming that the constructors actually use the values passed to them.

2. Another method which could be adopted is to compare the parameter types of constructor and the types of private variables for that class. The assumption behind this technique is that constructors are used to populate/initiate private variables for an object. This technique, although, seems to work fine for realistically written code but it does put lot of constraints on the code which can be accepted as input as copying parameter values to private variables is not the only way to populate the private variables. Also, this strategy has the disadvantage of being useless in all other cases.

Information content of constructors based on parameter types

If each data type can be given a number for its information content then sum of the information content for the parameters can denote the information content for the constructor. Thus, a constructor with maximum information content will be a prudent choice.

Rules:

1. int has information content of 1.
2. Array has information content of one more than its type. For example,
   int[] has information content of 2.
3. Any class has the information content of the constructor with maximum
   information content. For example, consider
   Class A has constructors:
   1. (int) - IC = 1
   2. (int[], int) - IC = 3
   Then IC of type A = 3 (max(1,3))

Quantifying the IC for constructors of a class

Two problems are being solved here. First, selection of a constructor for the purpose of instantiating an object. Second, avoid recursive/infinite instantiation.

Input to this problem is a set of classes, each with at least one constructor (default or non-default constructor). Assuming that we have all the classes that are used we can conclude that there has to be at least one class which has a constructor whose parameters are integers or arrays of integers otherwise the only way to break the cycle of recursive instantiation would be to use the default constructor.

Figure 4.14 Pseudo code to select preferred constructor

```
create a set Closure//to store discovered classes
create a set Classes and instantiate with names of classes
create a map<class name, int> score//stores the IC of class
score.put(int, 1) //score of int data type is 1
while Classes != Closure
  do for each class
      do for each constructor
          do for each parameter
              do if score.get(<parameter_type>) != null
                      then score[constructor] =
                                score[constructor] +
                                score.get(<parameter_type>)+
                                order_of_array
                   else score[constructor] = -1
                        continue
             end do
          end do
     end do
     max = 0
     for each constructor
       do if (score[constructor] > max)
             then max = score[constructor]
        end do
        if max > 0
            then score.put(<class_name>, max)
                 select this constructor for this class
  end do
```

At this stage all the information needed to build desired output is accumulated.

Instantiating a variable of specified type

We need variables of various types to serve as formal parameters for constructor and method calls. At this point it is important to have a closer look at the class block, figure 4.20. In each class block there is a reference variable “v1” which will point to the object of class that we are interested in. In constructor block variable “v1” is instantiated with each available public constructor of the class. In method block, figure 4.19, each available public method is called multiple numbers of times with variable “v1”; this will happen every time v1 is instantiated with a constructor. This whole process is repeated to give enough data to Daikon.

It should be noticed that each constructor instantiation and method call block resides inside a block structure "{}", this is done to reduce variable name clashes. With this design variable v2 might be declared in every block without causing any clash in scope. For each block a count is kept to keep track of the variables declared. Thus, to instantiate a variable of specified type inputs given will be type details and required variable number and output should be code which will instantiate a variable v<x>, where <x> is the variable number, of correct type.

Figure 4.15 Pseudo code for instantiation of variable

```
Instantiate(varNum, type)
      out = type.name + "v" + varNum + ";\n"
      //call holds the instantiation line
      call = "v" + varNum + "=new " + type + "( "
      preferredConst = get preferred constructor of
                       "type"
      if (order == 0)
            then for( i = 0; i < <no. of parameters in
                                  preferredConst>; i++)
                  do nextVar = count
                     count++
                     out = out + Instantiate(nextVar,
                                       preferredConst[i])
                     if(i != preferredConst.length -1)
                        then call = call + "v" +
                                       nextVar + " ,"
                        else call = call + "v" +
                                       nextVar + ");\n"
                  end do
            out = out + call
      return out
```

Above figure details the pseudo code to develop code which will instantiate a variable "v<varNum> of specified type. To instantiate a variable of some type first the constructor to be used is retrieved and all the variables required for its parameter list are instantiated and supplied to the constructor. The base case for this process is when an integer has to be instantiated.

Figure 4.16 Instantiation of integer

```
InstantiateInt(varNum, order)
      if (order == 0)
            return "int v" + varNum + " =
                        random(-100, 100);"
```

When order is a positive number then an array should be instantiated. Details of pseudo code will not be presented as it will be quite obvious what to do once an example is provided and also because pseudo code will be messy.

Figure 4.17 Instantiation of integer of order 2

```
int[][] v2 = new int[random(0,50)][random(0,50)];
for(int i1 = 0; i1 < v2.length; i1++){
for(int i2 = 0; i2 < v2[i1].length; i2++){
v2[i1][i2] = random(-100,100);
}
}
```

In case of non-integer type constructor should be called within the loop, along with all the variable instantiations required for that constructor call.

A case may arise where inside an array object instantiation loop we require another array object, perhaps an integer array; in this case the loop variables will clash in scope. To solve this problem the loop variable is different at different levels of loop. At first level it will be “i” producing loop variables like “i1”, “i2” ... at second level it will be “ii” producing loop variables like “ii1”...

Constructor block

Constructor block has code which instantiates variable “v1” with a public constructor of the class. Once we are able to produce code to instantiate a variable of specified type, producing constructor block becomes fairly simple and intuitive.

Figure 4.18 Pseudo code for developing constructor block

```
constructorBlock(input)
      call = "v1 = new " + input.Name + "(";
      for ( i = 0; i < input.parameterList.length;
                                              i++)
          do varNum = count++
               out = out + instantiate(varNum,
                           input.parameterList[i])
               if ( i !=
                    input.parameterList.length-1)
                  then call =call+"v"+varNum+ ", "
                  else call = call + "v" + varNum
           end do
      call = call + ");\n"
      out = out + call + "}\n"
      count = 2
      return out
```

In the above pseudo code input is an object pointing to details of the constructor we are interested in. Variable count is used to keep track of number of variables that have been instantiated in the current block; it becomes useful to name the next variable to be instantiated.

Method block

Method block produces code to call a particular public method of the class. It is similar to constructor block except that there might be a return type specified for the method in which case a variable of return type should be made to receive the result of method call.

Figure 4.19 Pseudo code for method block

```
methodBlock(input)
      out = "{\n"
      call = "v1." input.methodName + "(";
      for( i = 0 ; i < input.parameterList.length;
                                              i++)
         do  varNum = count++
            out = out + instantiate(varNum,
                      input.parameterList[i])
            if (i != input.paramterList.length-1)
               then call = call +"v"+varNum+ ", "
               else call = call + "v" + varNum
        end do
     call = call + ");\n"
     ref = declare(varNum, input.returnType)
     out = out + ref;
     out = out+ "v" + varNum + "=" + call  + "}\n"
     return out
```

Above pseudo code describes the working of methodBlock() method which accepts Method details as input and produces code to call that method. Functions of variables call, out, and count is same as in constructor block. declar(varNum, input.returnType) returns code which has a variable "v<varNum>" declared to be of type input.returnType, i.e., the return type for the method. In case a method does not have a return type then last part of method block is skipped.

Class block

Class block contains code to instantiate variable "v1" with every public constructor and then calling all public methods with the instantiated object. Producing this code using constructor block and method block is simple as it is just calling both methods for each constructor and method of the class.

Figure 4.20 Pseudo code for class block

```
classBlock(input)
      out = "{\n"
      out = declare(1, input.className)
      out = out + "for (int cIter = 0; cIter <
           <no. of constructors> ; cIter++){\n"
      out = out + "switch(cIter){\n"
      for each constructor
         do out = out +
       constructorBlock(input.constructorList[i])
         end do
      out = out + "}\n" // close class switch
                        // block
      out = out + "for (int mIter = 0; mIter <
           50*<no. of methods> ; mIter++){\n"
      out = out + "int select = random(0, <no. of
                                    methods>);\n"
      out = out + "switch(select){\n"
      for each method
            do out = out +
     "methodBlock(input.methodList[select]);\n"
            end do
      out = out + "}\n}\n}\n"
```

Above pseudo code takes class details as input. To print the whole code class blocks for each class is printed one after the other. This whole collection is kept inside a loop to repeat the process.

Figure 4.21 Output of random test generator for A.java and B.java

```
import java.util.Random;
public class HasMain {
public static int random(int Min, int Max){
Random rand = new Random(System.nanoTime());
int size = (Math.abs(Min) + Math.abs(Max));
int result = (Min + rand.nextInt(size));
return result;
}
public static void main(String[] args){
for(int loop=0; loop < 200; loop++){
{
A v1 = null;
for(int cIter = 0; cIter < 3; cIter++){
switch(cIter){
case 0:
{
v1 = new A();
}
break;
case 1:
{
int v2;
v2 = random(-100, 100);
int[] v3 = new int[random(0,50)];
for(int i1 = 0; i1 < v3.length; i1++) {
v3[i1] = random(-100, 100);
}
v1 = new A(v2, v3);
}
break;
case 2:
{
int[] v2 = new int[random(0,50)];
for(int i1 = 0; i1 < v2.length; i1++) {
v2[i1] = random(-100, 100);
}
v1 = new A(v2);
}
break;
}
for(int mIter = 0; mIter < 200; mIter++){
int select = random(0, 3);
switch(select){
case 0:
{
v1.metA1( );
}
break;
case 1:
{
int v2;
v2 = v1.metA2( );
}
break;
```

Figure 4.22 Output for A.java and B.java (continued)

```
case 2:
{
int v2;
v2 = random(-100, 100);
int[] v3 = new int[random(0,50)];
for(int i1 = 0; i1 < v3.length; i1++) {
v3[i1] = random(-100, 100);
}
int[] v4;
v4 = v1.metA3( v2, v3);
}
break;
case 3:
{
int[] v2 = new int[random(0,50)];
for(int i1 = 0; i1 < v2.length; i1++) {
v2[i1] = random(-100, 100);
}
v1.metA4( v2);
}
break;
}
}
}
}
{
B v1 = null;
for(int cIter = 0; cIter < 2; cIter++){
switch(cIter){
case 0:
{
v1 = new B();
}
break;
case 1:
{
A[] v2 = new A[random(0,50)];
for(int i1 = 0; i1 < v2.length; i1++) {
int v3;
v3 = random(-100, 100);
int[] v4 = new int[random(0,50)];
for(int ii1 = 0; ii1 < v4.length; ii1++) {
v4[ii1] = random(-100, 100);
}
v2[i1] = new A( v3 ,v4);
}
int[][] v5 = new int[random(0,50)][random(0,50)];
for(int i1 = 0; i1 < v5.length; i1++) {
for(int i2 = 0; i2 < v5[i1].length; i2++) {
v5[i1][i2] = random(-100, 100);
}
}
v1 = new B(v2, v5);
}
break;
```

Figure 4.23 Output for A.java and B.java (continued)

```
}
for(int mIter = 0; mIter < 100; mIter++){
int select = random(0, 1);
switch(select){
case 0:
{
A v2;
int v3;
v3 = random(-100, 100);
int[] v4 = new int[random(0,50)];
for(int i1 = 0; i1 < v4.length; i1++) {
v4[i1] = random(-100, 100);
}
v2 = new A( v3 ,v4);
v1.metB1( v2);
}
break;
case 1:
{
A[] v2 = new A[random(0,50)];
for(int i1 = 0; i1 < v2.length; i1++) {
int v3;
v3 = random(-100, 100);
int[] v4 = new int[random(0,50)];
for(int ii1 = 0; ii1 < v4.length; ii1++) {
v4[ii1] = random(-100, 100);
}
v2[i1] = new A( v3 ,v4);
}
A[] v5;
v5 = v1.metB2( v2);
}
break;
}
}
}
}
}
}
}
```

Rest of the steps

Once the code which runs the input ADTs with random input is generated these classes are fed as input to Daikon which produces the likely invariants and annotates Java classes with JML annotations which imply the produced invariants. The annotated code is fed as input to ESC/Java2 and its output is stored. These files are filtered to check which invariants ESC/Java2 issues warning on and these invariants are marked with a hyphen (-) in the annotated java code. Filtering ESC/Java's output and marking the results in Java files is done using Python scripts.

Relation between number of samples, correct invariants, and spurious invariants reported

In the current implementation, which supplies Daikon with test cases which are randomly built, we need to study the effect of number of samples (or amount of data) given to Daikon and the quality of invariants which are reported. Quality of invariants can be quantified in number of ways but here we will be measuring it by number of correct invariants and number of spurious invariants inferred, note that spurious invariants does not include invariant which are implied by other invariants they just include invariants which are false for the reported context.

Following tests were conducted on a standard implementation of Stack data structure. Invariants are studied for the public methods as they reflect the behavior of ADT, number of sample is calculated as the average number of times a public method is called.

Table 4.1 Variation of quality of invariants with number of samples

Samples	Total Invariants	Spurious Invariants	Correct Invariants
10000	30	0	30
5000	27	1	26
4000	31	2	29
2000	32	3	29
1000	33	3	30
800	38	8	30
600	40	10	30
400	48	18	30
200	61	31	30
100	60	33	27
80	61	33	28
60	67	39	28
40	61	35	26
20	64	38	26
10	76	54	22
5	68	50	18

Figure 4.24 Number of samples vs. spurious invariants

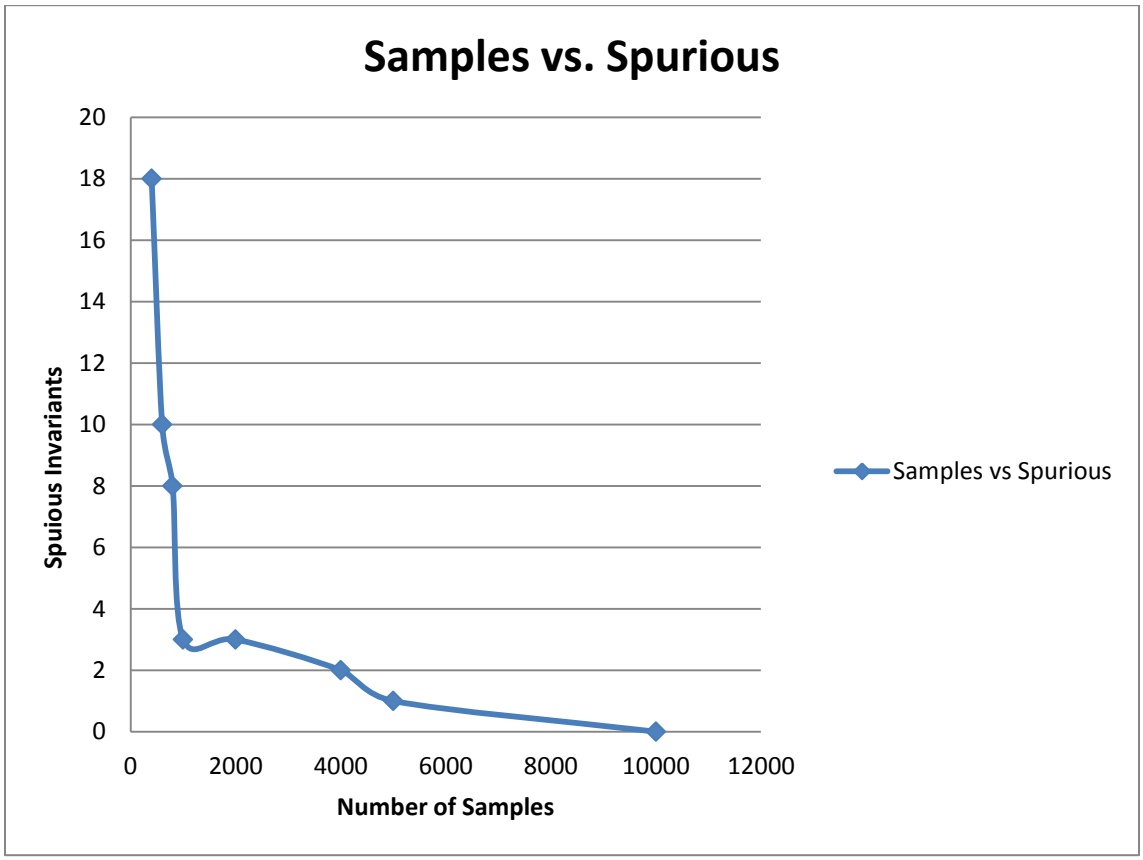
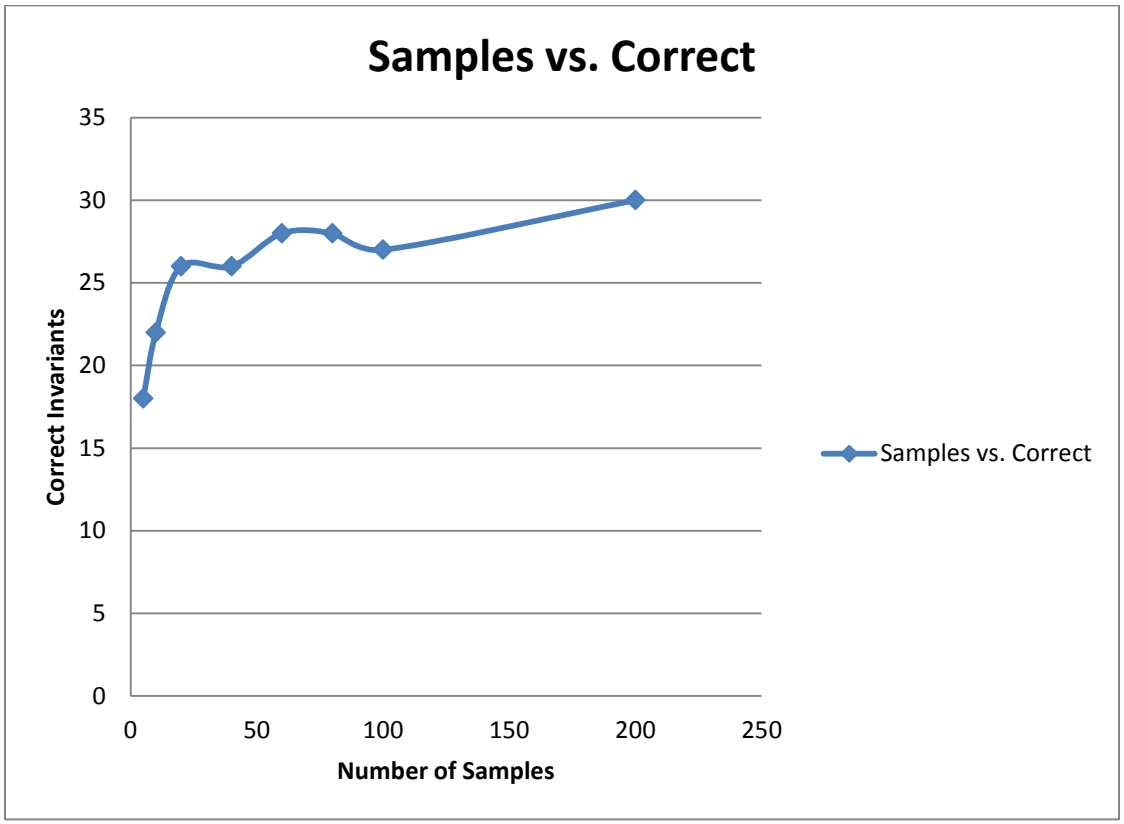


Table 4.1 provides the data collected with various experiments in which number of samples, which points to number of times a public method was executed, was changed and change in total number of invariants suggested, wrong or spurious invariants suggested, correct invariants suggested was recorded. Here invariants were observed over the public methods of Stack.java which are isEmpty(), push(int x), and pop(). It can be noticed that number of spurious invariants suggested increase with less number of samples, which is shown in Figure 4.24. Also, notice that number of correct invariants suggested increases with increase in number of samples, as suggested in Figure 4.25.

Figure 4.25 Number of samples vs. correct invariants



Chapter 5 - Results

This chapter presents some of the outputs of the implementation.

Sample Results

Stack Data Structure

Stack.java was implemented with following private variables, constructors, methods:

```
int[] S; //integer array used to store stack elements
int top; //points to the top of stack

Stack(); //Creates a stack of size 10
Stack(int x); //Creates a stack of size |x|

int isEmpty(); //returns 1 if stack is empty else returns 0
void push(int x); //if stack is not full then makes x as top of
                  //stack
int pop(); //if stack is not empty returns top element and
           //deletes the element else returns -1
```

Figure 5.1 Class invariants for Stack.java

```
/*@ invariant this.S != null; */
/*@ invariant this.top >= -1; */
/*----@ invariant daikon.Quant.size(this.S) >= 2; */
/*----@ invariant this.top <= daikon.Quant.size(this.S)-1; */
```

The class invariants obtained convey that the array used to store stack elements is never null and that value of top is always greater than -1. The last two invariants say that size of array S is always greater than 2, which is not always true, and that top value is always less than size of array S, both of which are warned about by ESC/Java2.

Figure 5.2 Invariants on constructors Stack() and Stack(int x)

```
Stack():
-@ ensures daikon.Quant.eltsEqual(this.S, 0);
@ ensures this.top == -1;
-@ ensures daikon.Quant.size(this.S) == 10;
Stack(int x):
-@ ensures daikon.Quant.eltsEqual(this.S, 0);
@ ensures this.top == -1;
-@ ensures this.top != \old(size);
-@ ensures \old(size) <= daikon.Quant.size(this.S);
```

Notice that ESC/Java complains about all the non-trivial invariants, which happen to be true.

Figure 5.3 Invariants for method isEmpty()

```
@ ensures this.S == \old(this.S);
-@ ensures daikon.Quant.pairwiseEqual(this.S, \old(this.S));
@ ensures this.top == \old(this.top);
@ ensures (this.top == -1)  <==>  (\result == 1);
@ ensures (this.top == -1)  <==>  (this.top < \result);
@ ensures (this.top >= 0)  <==>  (\result == 0);
@ ensures (this.top >= 0)  <==>  (this.top >= \result);
@ ensures \result == 0 || \result == 1;
@ ensures \result <= daikon.Quant.size(this.S)-1;
```

Notice that the implementation is able to produce invariants like stack is empty iff result is true (return value of 1), top = - 1 implies that stack is empty, and if stack is not empty iff result is false (return value of 0), top >= 0 implies that stack is not empty. Both the invariants are not warned by ESC/Java2.

Figure 5.4 Invariants for method push(int x)

```
@ public normal_behavior // Generated by Daikon
@ ensures this.S == \old(this.S);
-@ ensures daikon.Quant.size(this.S) ==
                        \old(daikon.Quant.size(this.S));
-@ ensures this.top >= 0;
@ ensures this.top >= \old(this.top);
-@ ensures \old(this.top) <= daikon.Quant.size(this.S)-1;
```

Notice that frame conditions reported are true and is proved by ESC/Java2. But, stronger invariants like top value is greater than or equal to old value of top is warned about by ESC/Java2.

Figure 5.5 Invariant for method pop()

```
@ ensures this.S == \old(this.S);
-@ ensures daikon.Quant.pairwiseEqual(this.S, \old(this.S));
@ ensures (\old(this.top) == -1)  ==>  (\result == -1);
@ ensures (\old(this.top) == -1)  ==>  (this.top == \result);
-@ ensures (\old(this.top) >= 0)  ==>  (\result ==
       daikon.Quant.getElement_int(this.S, \old(this.top)));
@ ensures (\old(this.top) >= 0)  ==>
                        (this.top - \old(this.top) + 1 == 0);
@ ensures this.top <= \old(this.top);
-@ ensures this.top < daikon.Quant.size(this.S)-1;
-@ ensures \old(this.top) <= daikon.Quant.size(this.S)-1;
```

Notice that frame conditions are correctly reported by Daikon but are warned about by ESC/Java2. Stronger invariants like stack is empty implies result is false is reported, also notice the presence of invariants which are implied by other invariants.

Queue Data Structure

Queue.java was implemented with following private variables, public constructors, and public method

int[] q; //integer array to hold elements of queue
int head; //points to the head of queue (dequeue)
int tail; //points to the tail of queue (enqueue)
int size; //holds the size of queue
Queue(int x); //Creates a queue of size |x| and initializes variables
int isEmpty(); //Returns 1 if empty else 0
int isFull(); //Returns 1 if full else 0
void enqueue(int x); //if queue is not full then puts x at tail
int dequeue(); //if empty returns -200 else deletes element at head and returns
// its value

Figure 5.6 Class invariants for Queue.java

```
*@ invariant this.q != null; */
/*@ invariant this.head >= 0; */
/*@ invariant this.tail >= 0; */
/*@ invariant this.size >= 0; */
/*--@ invariant this.head <= daikon.Quant.size(this.q); */
/*--@ invariant this.tail <= daikon.Quant.size(this.q); */
/*--@ invariant this.size <= daikon.Quant.size(this.q); */
```

Notice that the invariants suggest that the array “q” used to hold queue values is never null, and that values of head, tail, and size are always greater than or equal to zero. Last three

invariants suggest the relation between head, tail, size, and the size of array q, which are warned by ESC/Java2.

Figure 5.7 Invariants for Queue(int x)

```
@ public normal_behavior // Generated by Daikon
@ ensures this.head == this.tail;
@ ensures this.head == this.size;
-@ ensures daikon.Quant.eltsEqual(this.q, 0);
@ ensures this.head == 0;
@ ensures daikon.Quant.eltsEqual(this.q, this.head);
-@ ensures this.head != daikon.Quant.size(this.q)-1;
-@ ensures \old(size) <= daikon.Quant.size(this.q);
-@ ensures \old(size) != daikon.Quant.size(this.q)-1;
```

Invariants reported suggest that head, tail, and size have the value of zero, all values in array q are of zero, and that size of array q is greater than that of variable size. Few of these are warned about by ESC/Java, which should not occur as these invariants are correct.

Figure 5.8 Invariants for isEmpty()

```
@ ensures this.q == \old(this.q);
-@ ensures daikon.Quant.pairwiseEqual(this.q, \old(this.q));
@ ensures this.head == \old(this.head);
@ ensures this.tail == \old(this.tail);
@ ensures this.size == \old(this.size);
@ ensures (this.size == 0)  <==>  (\result == 1);
@ ensures (this.size == 0)  <==>  (this.size < \result);
-@ ensures (this.size == 0)  ==>  (this.head == this.tail);
@ ensures (this.size == 0)  ==>  (this.head >= this.size);
@ ensures (this.size == 0)  ==>  (this.tail >= this.size);
@ ensures \result == 0 || \result == 1;
```

Frame conditions reported are all true. Notice that invariants suggest that result is either 0 or 1 and size is 0 iff result is 1, other invariants about size being 0 and what it implies are true. One very important invariant is the fact that if size is zero then value of head is equal to that of tail, which is warned by ESC/Java2.

Figure 5.9 Invariants for isFull()

```
@ ensures this.q == \old(this.q);
-@ ensures daikon.Quant.pairwiseEqual(this.q, \old(this.q));
@ ensures this.head == \old(this.head);
@ ensures this.tail == \old(this.tail);
@ ensures this.size == \old(this.size);
-@ ensures (\result == 0)  <==>
           (this.size <= daikon.Quant.size(this.q)-1);
@ ensures (\result == 0)  ==>  (this.head >= \result);
@ ensures (\result == 0)  ==>  (this.size >= \result);
@ ensures (\result == 0)  ==>  (this.tail >= \result);
-@ ensures (\result == 1)  ==>  (this.head == this.tail);
@ ensures (\result == 1)  ==>  (this.head <= this.size);
@ ensures (\result == 1)  ==>  (this.tail <= this.size);
@ ensures \result == 0 || \result == 1;
```

Results are similar to that of isEmpty() method.

Figure 5.10 Invariants for enqueue(int x)

```
@ ensures this.q == \old(this.q);
@ ensures this.head == \old(this.head);
-@ ensures daikon.Quant.size(this.q) ==
                        \old(daikon.Quant.size(this.q));
@ ensures this.size >= \old(this.size);
-@ ensures \old(this.tail) <= daikon.Quant.size(this.q);
-@ ensures \old(this.size) <= daikon.Quant.size(this.q);
```

Again frame condition are reported correctly. Also, it is inferred correctly that value of head does not change, which is changed only in dequeue operation.

Figure 5.11 Invariants for dequeue()

```
@ ensures this.q == \old(this.q);
-@ ensures daikon.Quant.pairwiseEqual(this.q, \old(this.q));
@ ensures this.tail == \old(this.tail);
-@ ensures (-@ ensures (\result == -200)  <==>
                            (this.head == \old(this.head));
-@ ensures (\result == -200)  <==>
                            (this.size == \old(this.size));
@ ensures (\result == -200)  <==>  (\old(this.size) == 0);
-@ ensures (\result == -200)  ==>  (this.head == this.tail);
@ ensures this.size <= \old(this.size);
@ ensures \old(this.head) <= daikon.Quant.size(this.q);
@ ensures \old(this.size) <= daikon.Quant.size(this.q);
```

Frame conditions are reported correctly. It is inferred correctly that value of tail does not change, also it is inferred correctly that when -200 is returned as result then the queue is empty.

Other interesting invariants

Figure 5.12 Class invariant for Binary Search Tree

```
/*@ invariant this.root.key > this.root.l.key; */
/*@ invariant this.root.key <= this.root.r.key;*/
/*@ invariant this.root.l.key < this.root.r.key; */
```

The implementation correctly infers the important property that value of key of root is greater than the key value of its left node and is less than or equal to the value of key of its right node. ESC/Java2 timed out while trying to check this and thus did not complain about the results.

Analysis

From the results it can be seen that frame conditions are being correctly inferred and no preconditions are being inferred, which can be expected as the test cases are given random valued parameters. ESC/Java's performance is not consistent as it warns about invariants which are correct and also about invariants which are incorrect, which stems from limitations of ESC/Java's implementation. Also, whenever there is any loop involved ESC/Java is not able to prove any properties about that code. Daikon produces large number of invariants some of which are implied by invariants which are already reported.

Chapter 6 - Conclusion and Future Work

Conclusion

This project describes a method to automatically generate program contracts using Daikon and increases the confidence of generated contracts using ESC/Java2. The random test case generator developed does not guarantee code coverage, but tuning it to generate large number of test cases produced strong and usable results on tested data structures. Contracts produced describe post conditions and frame conditions for each method and class invariants. Statically checking the suggested invariants using ESC/Java2 was less useful than expected because of strong assumptions about code which are made by ESC/Java, which resulted in high number of false negatives. The only way to efficiently use ESC/Java2's capabilities is to annotate the code manually and evolve the annotations based on output of ESC/Java2, and it is not a good idea to use it to check automatically generated invariants without any user input.

Future Work

1. Extend tool to check all Java primitive data types: Current implementation checks only those Java classes which use integer data types, integer arrays, other classes which are checked, or arrays of these classes. This restricts the usability of the developed tool. This can be solved by including checking for other Java primitive data types.

2. Include other Java features: Current implementation restricts classes to have only fields, methods, and constructors. To extend this tool to check all Java programs it is required to check all features offered by Java.

3. Experiment with other static checkers: Although ESC/Java2 is a powerful static checker it makes very strong assumptions about Java which results in warning of many of the correct invariants suggested by Daikon. A static checker which results in less number of false warnings is required.

4. Generate test cases which ensure code coverage: Current random test case generator does not guarantee code coverage which forces manual inspection of invariants generated to check if behavior of program is conveyed properly. The reliance on manual inspection can be reduced if test cases can guarantee code coverage.

References

[1] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants.*Sci. Comput. Program.* 69, 1-3 (December 2007), 35-45.

[2] Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In*Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '02). ACM, New York, NY, USA, 191-202.

[3] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying loop invariant generation using splitter predicates. In *Proceedings of the 23rd international conference on Computer aided verification* (CAV'11), Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer-Verlag, Berlin, Heidelberg, 703-719.

[4] Laura Kovacs and Andrei Voronkov. 2009. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (FASE '09), Marsha Chechik and Martin Wirsing (Eds.). Springer-Verlag, Berlin, Heidelberg, 470-485.

[5] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering* (ICSE '99). ACM, New York, NY, USA, 213-224.

[6] MIT Program Analysis Group. The Daikon invariant detection user manual. http://groups.csail.mit.edu/pag/daikon/.

[7] Frank Groeneveld. Daikon javascript output. https://github.com/frenkel/daikon-javascript-output/blob/master/java/daikon/Quant.java.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (PLDI '02). ACM, New York, NY, USA, 234-245.

[9] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Tech. Note 2000-002, Compaq SRC, Oct. 2000.

[10] Jeremy W. Nimmer and Michael Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java, 2001.