ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 13, April 2016

A Framework for Automatic Checking Loop Invariants using Mutation, Dynamic Analysis and Static Checking

P.Sasidhar reddy[1,] C.Silpa[2]
psasi580@gmail.com silpa.c8@gmail.com

[1]Department of IT, Sree Vidyanikethan Engineering College, Tirupati, India
[2]Department of IT, Sree Vidyanikethan Engineering College, Tirupathi, India

Abstract:Automatic program verification, proving program correct still requires substantial expert manual effort. One of the biggest burden is providing loop invariants properties that hold for every iteration of a loop. Compared to other requirement elements such as pre -and post conditions, loop invariants tend to be difficult to understand and to express. The proposed system automates the functional verification of incomplete correctness of programs with loops by inferring the required loop invariants. In this approach it combines complementary techniques such as test case generation, dynamic invariant detection, and static verification. This approach can be implemented by a tool called DYNAMATE. DYNAMATE improves the flexibility of loop invariant inference by combining static (proving) and dynamic (testing) techniques. The DYNAMATE tool presented in this process combines different techniques with the overall goal of providing fully automatic verification of programs with loops

Keywords– Loop Invariants, Mutation testing,, Dynamic analysis, Static checking, Dynamate

1. Introduction

Verifiers that can confirm programs correct against their full functional specification require, for programs with loops extra annotations in the form of loop invariants. For programs with loops, one of the biggest burdens is providing loop invariants property that hold for every iteration of a loop Compared to pre- and post conditions, it is much more difficult to write loop invariants, In this approach evolution automation of full program verification through loop invariants[1]. This approach is based on included of static (program proving) and dynamic(testing) techniques The current DynaMate prototype combine the EvoSuite[6] test case generator, the Daikon invariant detector[7] and the ESC/Java2 static verifier[8]. Fully automatic verifiers such as cccheck or BLAST fail to establish the correctness of the annotated program., auto-active verifiers such as ESC/Java2 succeed,

In Exiting system Verifiers that can confirm programs correct against their full functional specification require programs with loopsPrograms with loops, one of the main burdens is providing loop invariants properties that hold for every iteration of a loop. this mainly drawback Loop invariants should be complicated to analyze. The proposed system automates the functional verification of partial truth of programs with loops by inferring the required loop invariantsIn this approach it combines complementary techniques such as test case generation, dynamic invariant detection, and static verificationThis approach can be implement by a tool called DYNAMATE, a fully automatic verifier for Java programs with loops. DYNAMATE improves the flexibility of loop invariant inference by integrate static (proving) and dynamic (testing) techniques this advantage of Identify Loop invariants program with easy analysis Dynamate is best performance with other tool. our DYNAMATE prototype automatically discharged 97% of all proof program, resulting in automatic complete correctness proofs of 25 out of the 28 methods—outperforming a number of state-of-the-art tools for fully automatic verification.

1.1Evo suite

This new approach in the EVOSUITE tool, and compared it to the common approach of addressing one goal at a time. Evaluated on open source librariesThe EVOSUITE tool implements the approach presented ingenerating JUnit test suites for Java code.EVOSUITE works on the byte-code level

and collect allnecessary information for the test cluster from the byte-codevia Java indicationDuring test generation, EVOSUITE considers one toplevel class at a time. The class and all its unnamed and member classes are instrumented at byte-code level to keep track of called methods and branch distances during execution. To produce test cases as compliable JUnit source code, EVOSUITE accesses only the public interfaces for test generation; any subclasses are also careful part of the unit under test to allow testing of abstract classes. To execute the tests throughout the search, EVOSUITE uses Java Reflection.

This technique to automate test generation. shown that optimizing whole test suites toward a coverage criterion is superior to the traditional approach of targeting one coverage goal at a FRASER AND ARCURI: WHOLE TEST SUITE GENERATION time. In our experiments, this results in significantly betteroverall coverage with smaller test suites.

1.1.2 Diakon

Daikon is an execution of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. An invariant is a assetsthat holds at a certain point or points in a program; these are often seen in declare statements, documentation, and formal specifications.Dynamic invariant detection runs a program,

All Rights Reserved © 2016 IJARBEST

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 13, April 2016

observes the values that the program computes,and then information properties that were true over the observed executions. Daikon can detect property in C, C++, C#, Eiffel, F#, Java, Perl, and Visual Basic programs
Shortcut for the impatient: skip directly to the fitting instructions
for[Unix/Linux/MacOSXinstallation,Windows installation]

This section gives gradually instructions for installing Daikon.Here is an summary of the steps. Details appear below;

select the instructions for youroperating system.

1. Download Daikon.
2. Place three commands in your shell initialization file.
3. Optionally, modify your installation.
4. Optionally, compile Daikon and construct other tools.

Requirements for running Daikon In order to run Daikon, you must have a Java 7 (or later) JVM (Java Virtual Machine).You must also have a Java 7 (or later) compiler

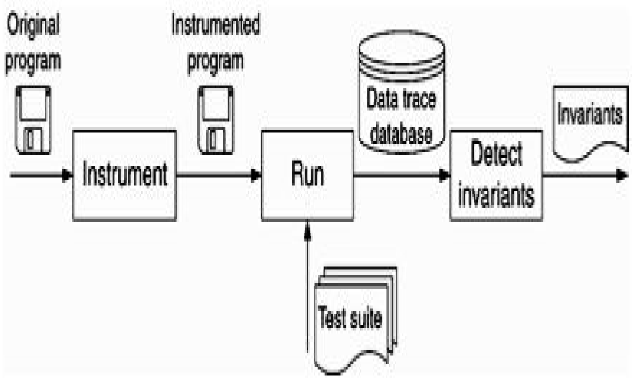

Fig 1 :Daikon's Infrastructure

Daikon proposes to automatically determine program invariants and report them in a meaningful manner

Original Program
I,s=0,0print b,n
Do i!=n
I=i+1,s=s+b[i]
od do i!=n

Instrumented Program
I,s =0,0
print I,s,n,b[i]

i=i+1,s=s+b[i]
do

Trace File

```
15.1.1:::ENTER
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified
I = 0, modified
s = 0, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 1, modified
S = 92, modified
```

Invariants
1.) n >= 0
2.) s = SUM(B)
3.) i >= 0

1.1.3 ESC/java2

ESC/Java2 is a tool for staticverification program specifications. It expandssignificantly upon ESC/Java, on which it is built. It is reliable with the definition of JMLand of Java 1.4. It adds additional static checking to that in ESC/Java; most considerably, itadds support for checking frame conditions and annotations containing method calls. Thisdocument describes the position of the final release of ESC/Java2, along with some notesregarding the details of that implementation
JML should be easy to use for any Java programmer
JML assertions are added as comments in .java file, between /*@ . . . @*/, or after //@,Properties are specified as Java boolean expressions, extended with a few operators (\old, \forall, \result,. . .). using a a small number of keywords (requires, ensures, signals, assignable, pure, invariant, non null, . . .)
The goal of the ESC/Java2 work is to expand the use of ESC/Java by
a. updating the parser of ESC/Java so that it is consistent with the present definition of JML and Java,
b. packaging the updated tool so that it is more easily available to a big set of users, consistent with the source code license provisions of the ESC/Java source code,
c. and extending the choice of JML annotations that can be checked by the tool, where possible and where consistent with the engineering goals of ESC/Java.
the status of their implementation in ESC/Java2, the degree to which the annotation is logically checked, and any differences between ESC/Java2and JML.

1.2 overview of the dynamate

DYNAMATE inputs a program M and its specification—a precondition P and a post condition Q. Two outcomes of the algorithm are possible: achievement means that DYNAMATE has found a set of valid loop invariants that are sufficient to statically verify M beside its specification (P,Q); failure means that DYNAMATE cannot find new valid loop invariants, and those found are insufficient for static

All Rights Reserved © 2016 IJARBEST

verification. DYNAMATE's main loop starts by executing the test case generator, which produces a new set T. of test cases that implement M with inputs satisfying the precondition P. The loop feds on the whole set TS of test cases generated so far to the dynamic invariant detector, which outputs a set of candidate loop invariants I To find out which candidates are indeed valid, DYNAMATE calls the static verifier on the program annotated with all candidates I the verifier income a set of proved candidates J (a subset of I), which DYNAMATE adds to the set Inv of established loop invariants. Then, using the current Inv, it calls the static verifier again, this time trying a full rightness proof of M against (P,Q). If verification succeeds, DYNAMATE terminates with success a static verifier that is sound but incomplete, unproved candidates in I n INV are not necessarily invalid.

Algorithm: dynamate
Require: program M, precondition p,postcondtionq,
TS (set of test case)
INV (set of verified loop invariants
C (set of candidate)
While static verification can't prove (M,P,Q,INV)
T ← execute test case generator on (M,TS)
If I has not changed then
Return ("failure",IVN)
End if
M'← annotate M with candidate invariants I
J ←statically check valid invariants of (M',P)
INV ← INV U J
C ←I\INV
End while
Return ("success",INV)

1.2.1Running Example: Binary Search
binarySearch0, a helper method declared in class java.util.Arrays in the standard Java

```
private static int binarySearch0(int[] a,intfromIndex,int toIndex,int key)
    {
        int low = fromIndex, high = toIndex - 1;
        while (low <= high)
        {
          // midpoint of [low..high]
            int mid = low + ((high -low)/2);
            int midVal = a[mid];
            if (midVal < key)
           low = mid+1;
          else if (midVal > key)
           high = mid - 1;
          else return mid; // key found
        }
            return -(low + 1); // key not found
    }
```

Fig. 2. Binary search method in java.util.Arrays

Fig. 1 shows binarySearch0, a helper method declared inclass java.util.Arrays in the standard Java API.

```
. /*@
   @ requires a != null;
@ requires TArrays.within(a, fromIndex, toIndex);
 @ requires TArrays.sorted(a, fromIndex, toIndex);
   @
   @ ensures \result _=>0 =) a[\result] = key;
   @ ensures \result < 0
   @ =) :TArrays.has(a, fromIndex, toIndex,
key);
  @*/
```

Fig. 3. Pre- and postcondition of binarySearch0.

Fig. 1 shows binarySearch0, a helper method declared in class java.util.Arrays in the standard Java API.

```
/*@
  @ loop_invariant fromIndex _ low
  @ loop_invariant low <= high + 1
  @ loop_invariant high < toIndex
  @ loop_invariant
:TArrays.has(a,fromIndex,low,key)
  @ loop_invariant
:TArrays.has(a,high+1,toIndex,key)
 @*
```

Fig. 4. Loop invariants required for verifying method binarySearch0

JML [2], using model-based predicates [3], representing implicit quantified expressions, with descriptive names. For example, the condition :TArrays.has(a, fromIndex,toIndex, key) means that array a has no element key over the interval range from fromIndex (included) to toIndex (excluded).

2 .Related work

DYNAMATE is center this section on the problem of inferring loop invariants to automate functional verification

2.1 Integrating Diakon and ESC/java

Dynamic detection propose likely invariants based on program executions, but the resulting properties are not guaranteed to be true of over all possible executions Static verification checks that properties are always true, but it can be difficult and dull to select a goal and to annotate programs for input to a static checker. Combining these techniques overcomes the weakness of each. how to integrate two

All Rights Reserved © 2016 IJARBEST

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 13, April 2016

complementary techniques for manipulating program invariants: dynamic invariants detection and static verification[74] Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java

2.2 Identifying loop invariants

Identifying for invariants using genetic programming and mutation testing[80] As most programs are not annotated with invariants, before research has attempted to automatically produce them from source code In this new approach to invariant generation using search. reuse the test generation front-end of existing tool Daikon and integrate it with genetic programming and a mutation testing tool There are two exceptional problems to be solved : firstly, to reduce the number of uninteresting invariants produced and secondly, to show the search to invariants that may be interesting but deceptive" to the search

2.3 verification java program

proof of Java programs using symbolic execution and invariant generation[9] Software verification is recognized as an impart and complicated problem Presented a novel framework based on symbolic execution , for the verification of software This framework explanation in the from of technique specification and loop invariants. Our framework is built on top of the java path finder form checking toolset and it was used for the verification several non-trivial java program

2.4 static techniques

Combination of static techniques. HAVOC using a static verifier to check if candidate assertions are valid: it creates an early set of candidates (possibly including loop invariants) by applying a fixed set of rules to the available component-level contract (i.e. component, invariants and interface specifications). Like DYNAMATE, HAVOC[12] applies the HOUDINI algorithm to establish which candidates are valid. Using only static techniques

2.5 Dynamic Techniques

The GUESS-AND-CHECK [13] algorithm infers invariants in the form of algebraic equalities (polynomials up to a given degree) The GUESS-AND-CHECK algorithm proceeds iteratively in two phases The “guess” phase uses linear algebra techniques to competently derive a candidate invariant from data. This candidate invariant is subsequently validated in a “check” phase dynamical discovery invariants instrumental techniques While the overall structure of GUESS-AND CHECK has some similarities to ours, DYNAMATE targets general-purpose programs, which requires very different techniques. The work on DAIKON [7]

2.6 Hybrid Techniques

CEGAR techniques has combined static verification and test case generation. The SYNERGY algorithm [14] .The DASH algorithm builds on SYNERGY to handle programs with pointers without whole-program may-alias analysis Two broad approaches to property checking are testing and verification Testing works best when errors are easy to find, but it is often difficult to get sufficient coverage for correct programs verification methods are most successful when proofs are easy to find, but they are often incompetent at discovering errors.

3.How Dynamate work

The program code is first fedinto a test case generator , which generates executions covering official behavior. From these, two dynamic invariant detector tools mine possible loop invariants, based both on fixed patterns (DAIKON)[7] as well as post conditions (GIN-DYN)[5] The candidates are not invalidated by the generated runs and then fed into a symbolic program verifier The verifier then may create a program proof (bottom right), but may also disprove candidates, which initiates another round of executions, and thus developed invariants

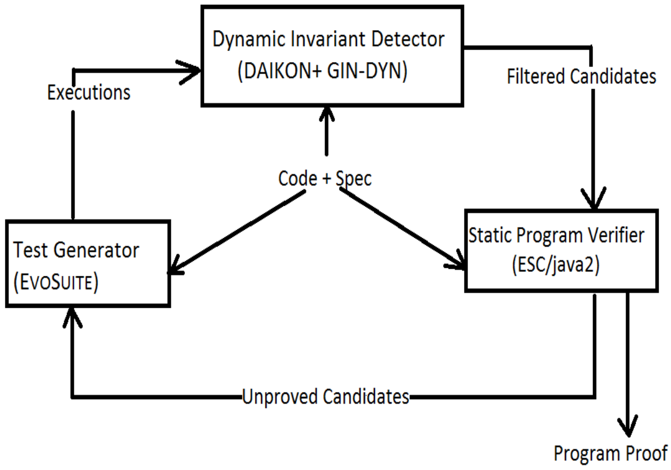

Fig 5 :Dynamate work

If the verifier fails to verify the program correct, a round of four steps begins

Step 1: test cases: To carry dynamic invariant detection, a test case generator construct executions of the program that satisfy the given precondition.

Step 2: candidate invariants. From the resulting executions, an invariant detector animatedly mines candidates for loop invariants.

Step 3: invariant verification. The existing set of loop invariant candidates are fed into a static program verifier.

Step 4: program verification and modification. Using the verified invariants, the static verifier may also be competent to produce a proof that the program is accurate with respect to its specification. If the proof does fail using the loop invariants inferred so far, another round generating, mining, and verifying starts.

How DYNAMATE works, using binarySearch0 as running example

3.1 Input: Programs and Specifications

DYNAMATE receives as input a Java method M with its functional specification consisting of precondition P and postcondition Q. Pre- and postcondition are written in JML.

All Rights Reserved © 2016 IJARBEST

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 13, April 2016

P and Q generally consist of a number of clauses, each denoted by the keyword **requires** (precondition) and **ensures** (postcondition) . While DYNAMATE can work with JML specifications in any form, to find it efficient to follow the principlesof the model-based approach to specification
Following the model-based specification style entails three main advantages for our work. First, it improves the abstraction and clarity of specifications, and hence it also facilitates reuse with dissimilar implementationsit should be clear that has(a, fromIndex, toIndex, key) means that array a contain a value key within fromIndex and toIndex.

Second, model-based specifications also make it easy to resolve static and a runtime semantics. When developing predicates in TArrays we defined each predicate as a **static boolean** method with both a Java implementation and a JML specification

Fromindex<= I <=toindex^key=a[i];

A third advantage of using model-based specifications is leveraged by the DYNAMATE approach and more precisely by the GIN-DYN invariant detector described.

3.2Test Case Generation

The DYNAMATE algorithm needs tangible executions to dynamically gather loop invariants DAIKON mines relations that hold in all passing test cases and GIN-DYN filters out invalid loop invariant candidates that are inaccurate by a test case While any test case generator could work with DYNAMATE, our prototype integrates EVOSUITE [6], a completely automatic search-based tool using a inherent algorithm
Since EVOSUITE tries to maximize branch coverage, it has a good chance of produce tests that pass all precondition checks and thus represent valid executions according to the specification

3.3 Dynamic Loop Invariant Inference

The DYNAMATE algorithm lies a component that detects "likely" loop invariants based on the actual executions provided by the test case generator. The present DYNAMATE implementation in two modules with balancing functionalities.
DAIKON's and GIN-DYN's invariants are complementary; for example, neither one suffices for a correctness proof of binarySearch0. DAIKON invariants are usually an essential basis to establish GIN-DYN
How DYNAMATE uses GIN-DYN and DAIKON.

Dynamic invariants detection with DAIKON

DAIKON [7] is a broadly used dynamic invariant detector which supports a set of basic invariant templates. Given a test suite and a set of program locations as input, DAIKON instantiates its templates with program variables, and traces their values at the locations in each and every one executions of the tests.
Since DYNAMATE needs loop invariants, it instructs DAIKON to draw variables at four different location of each loop: before loop entry, at loop entry, at loop exit, and after loop exit.

TABLE 1
loop invariant candidates produced by DAIKON in the first iteration of DYNAMATE.

ID	CANDIDATE	VALID
c_1	$\texttt{key} \in \{-1030, 0\}$	NO
c_2	$\texttt{a} \neq \textbf{null}$	YES
c_3	$\texttt{a[]}$'s elements one of $\{-915, 0\}$	NO
c_4	$\texttt{Arrays.INSERTION\_THRESHOLD} \neq \texttt{toIndex}$	NO
c_5	$\texttt{low} \geq \texttt{fromIndex}$	YES
c_6	$\texttt{low} \geq \texttt{key}$	NO
c_7	$\texttt{high} < \texttt{toIndex}$	YES
c_8	$\texttt{high} > \texttt{key}$	NO
c_9	$\texttt{high} \leq \texttt{a.length} - 1$	YES
c_{10}	$\texttt{toIndex} > \texttt{fromIndex}$	NO

GIN-DYN: Invariants from Postconditions

GIN-DYN: a way to efficiently generate a large amount of invalid or uninteresting invariant candidates how GINDYN does the filtering, again based on a mixture of dynamic and static techniques. The relax of the current section briefly discusses how invariant candidates formed by GIN-DYN are used within DYNAMATE. In truth, GIN-DYN produces the two fundamental invariants on lines in Figure 4 necessary for a truth proof of binarySearch0. The final set of verified loop invariants includes those of with 28 more, consisting of 13 invariants establish by DAIKON and 20 invariants found by GIN-DYN.

3.3Static Program Verification

The DYNAMATE algorithm complement dynamic analysis with a static program verifier, which serves two purposes: (1) verifying loop invariant candidates, and (2) using verified loop invariants to carry out a conclusive truth proof.

proof of Loop Invariants

The DYNAMATE prototype relies on the ESC/Java2 static verifier, which works on Java programs and JML annotations.
DYNAMATE always calls ESC/Java2 with the –loopSafe option enabled.

Program Proof

At the end of each iteration, DYNAMATE uses the present set of valid loop invariants to attempt a correctness proof of the program beside its specification. If ESC/Java2 succeeds, the whole DYNAMATE algorithm stops with success

Refining the Search for Loop Invariants

Original loop invariant candidates may be overspecificand hence unsound— Since this may indicateunknown program behavior,for every such candidate L, DYNAMATE adds the conditional check

3.4 Experimental result

DYNAMATE automatically verified 25 of the 28 subjects, with high repeatability. On average, 66% of DYNAMATE's implementation time isorganization EVOSUITE, 15% in GIN-DYN, 14% in ESC/Java2 and 6% in DAIKON. DYNAMATE's average running time per method (45 minutes) is high 14 compared to other dynamic techniques. There are ample margins to optimize the DYNAMATE prototype for better speed;

DYNAMATE in action on the implementation of binary search available in class java.util.Arrays from Java's JDK.

1 /*@ **requires** a ! = **null**

All Rights Reserved © 2016 IJARBEST

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 13, April 2016

2 @ **requires** TArrays.within(a, fromIndex, toIndex)
3 @ **requires** TArrays.sorted(a, fromIndex, toIndex);
4 @ **ensures** \result≥0 => a[\result] = key;
5 @ **ensures** \result < 0 =>:TArrays.has(a, fromIndex, toIndex, key); @*/
6 **private static int** binarySearch0(**int**[] a, **int** fromIndex, **int** toIndex, **int** key)

Fig. 6: JML specification of the binary search method from java.util. The specificationincludes a precondition (**requires**) and two postconditions (**ensures**)

DYNAMATE mutates its influence and checks if any of the mutations are loop invariants. Among many mutations, :has(a, fromIndex, low, key) and :has(a, high + 1, toIndex, key) are valid loop invariants, essential to establishing the postcondition. DYNAMATE finds them during iteration # 9, validates them, and uses them to prove the second postcondition. This concludes DYNAMATE's run, which finishs successfully having achieved full verification

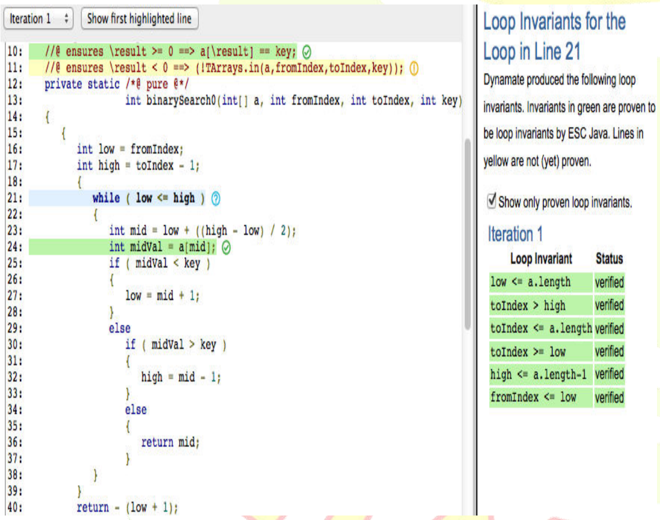

Fig. 6: DYNAMATE's report after iteration # 1 on binarySearch0. Verified statements and annotations (first and last highlighted element) are shown in green, unverified ones in yellow. Loop headers are highlighted in light blue. The right frame shows the proven loop invariants for the selected loop.

7 fromIndex ≤ low ^ low _≤high + 1 ^ high < toIndex
8 :TArrays.has(a,fromIndex,low,key)
9 :TArrays.has(a,high+1,toIndex,key)

Fig. 6: Loop invariants inferred by DYNAMATE

Our implementation Esc/java2 to identify the defects in programs with loop

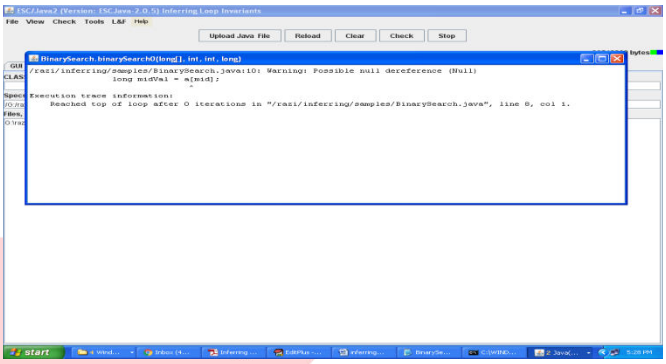

Fig 7. Identify the defects

After analyzing esc/java2 is giving above problems in codeNow click on 'Results' tab in main window to view tree

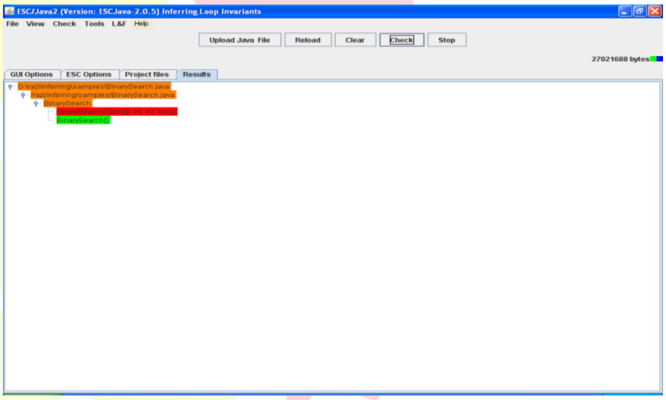

Fig 8. Results

White: the item has not been processed

Red: An error or static checker warning occurred in processing

Green: All checker passed

Yellow: some caution were generated, but no error warning

Orange: some child nodes have errors.

Blue: the static checker timed out or the verification condition was too larg

3.5Experimental Comparison

A case study that useful DYNAMATE to 28 methods from the java.util classes in the Java set library, including the binarySearch03 method. DYNAMATE automatically discovered allloop invariants in Figure 4 given the code and specification in Figure 2 and Figure 3, resulting in fully automatic verification of the binarySearch0 method.whole case study, DYNAMATE discharge 97% of the proof obligations of all the methods, resulting in full truth proofs for 25 of the 28 methods.

All Rights Reserved © 2016 IJARBEST

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 13, April 2016

TOOL	PROOF OBLIGATIONS EXPRESSIBLE	PROOF OBLIGATIONS PROVED	VERIFIED METHODS	TIME
ESC/Java2	100 %	231 (63 %)	0/28	122 s
INVGEN	42 %	60 (39 %)	0/28	78 s
BLAST	100 %	238 (65 %)	3/28	5431 s
cccheck	100 %	276 (75 %)	3/28	106 s
DYNAMATE	100 %	354 (97 %)	25/28	75348 s

DYNAMATE's evaluation (including the specification conditions in TArrays and TLists) into a form open to to each tool: to C for INVGEN and BLAST, and to C# forcccheck.
DYNAMATE achieves a solid 97% of automatically dischargedproof obligations, a lot improving over the state of the art: the proof compulsions discharged by DYNAMATE are a superset of those checked by other tools.9 In particular, DYNAMATE achieve full verification of 25 out of 28 methods, while the other tools established at most 3 methods. DYNAMATE automatically confirmed 28% more proof obligations than state-of-the-art verification tools.
A part of future work is evaluating DYNAMATE on examples initially used to evaluate INVGEN, BLAST, or cccheck; and integrate in DYNAMATE other intermediary tools modified to some kinds of invariants.

Other tools. The following table summarizes the critical features that distinguish DYNAMATE from a a small number of other cutting-edge tools

Tool	A/U	Limitation
Javapathfinder[9]	A	bounded symbolic execution
Vampair [10]	U	linear array access, no nesting
Srivastava.et.al [11]	U	requires templates and prediction

4. Summary of Contribution

The main analysis of this paper are:
1) DYNAMATE: an algorithm to automatically discharge proof obligations for programs with loops, based on a grouping of dynamic and static techniques.
2) GIN-DYN: an automatic technique to increase the dynamic detection of loop invariants, based on the idea of syntactically mutating postconditions [8].
3) This implementation of the DYNAMATE algorithm that integrates the EVOSUITE test case generator, the DAIKON dynamic invariant detector, and the ESC/Java2 static verifier, as well asGIN-DYN.
4) An evaluation of our DYNAMATE prototype on acase study linking 28 methods with loops fromjava.util classes.
5) A comparison against state-of-the-art tools forautomatic verification based on predicate abstract

5. Conclusion and Future work

This problem overcomes three techniques and used test case generated, dynamic invariants detection, and static verification this three techniques as development our prototype Dynamate automatically discharged 97 percent of all proof obligations for 28methodswith loops from java.util classes. Then esc/java2 frame is identify the defects in loop and methods

Our future work will focus on the following issues:

Better test generators. As any module in DYNAMATE can be replaced by a better implementation of the same functionalities, currently investigating dynamic/symbolic approaches to test case generation [16] as well as hybrid techniques integrating search-based and symbolic approaches [17].

More diverse invariant generators

This techniques based on symbolic execution such as the one implemented in DYSY [19]to provide for more, and more diversified, loop invariant candidates.

Stronger component integration

DYNAMATE can become a platform on which several approaches to test generation, dynamic analysis, and static verification can work in synergy[14] to produce a greater whole

http://www.st.cs.uni-saarland.de/dynamate/

6. Reference

[1] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," ACM Comp. Sur., vol. 46, no. 3, p. Article 34, January 2014. 19

[2] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML:" a behavioral interface specification language for Java," ACM SIGSOFT Software Engineering Notes, vol. 31, no. 3, pp. 1–38, 2006.
[3]N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying reusable components," in VSTTE, ser. LNCS, vol. 6217. Springer, 2010, pp. 127–141
[4] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in FMCO, ser. LNCS. Springer, 2006, pp. 342–363.
[5] C. A. Furia and B. Meyer, "Inferring loop invariants Usingpostcondition in Fields of Logic and Computation, vol.6300. New York, NY, USA: Springer, 2010, pp. 277–300
[6]G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in Proc. 11th Int. Conf. Quality, 2011, pp. 31–40
[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin," Dynamically discovering likely program invariants tSupport program evolution," IEEE Trans. Softw. Eng., vol. 27, no. 2, pp. 99– 23, Feb. 2001.
[8] D. R. Cok, J. R. Kiniry, and D. Cochran. (2008, Oct.). ESC/Java2 implementation notes. Kind Softw., Tech . Rep.[Online]. Available: http://goo.gl/BFn1zh
[9] C. S. Pasareanu and W. Visser, "Verification of Java programs using symbolic execution and invariant generation," in Proc. 11th Int. SPIN Workshop, 2004, pp . 164–181.

All Rights Reserved © 2016 IJARBEST

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 13, April 2016

[10] K. Hoder, L. Kovács, and A. Voronkov, "Invariant generation in Vampire," in TACAS, ser. LNCS, vol. 6605. Springer, 2011, pp. 60–64.
[11] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in PLDI. ACM, 2009, pp. 223–234
[12] S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and TWies, "Intra-module inference," in Proc. Int. Conf.ComputAided Verification, 2009, pp. 493–508.
[13] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. Nori , "A data driven approach for algebraic loop invariants," inProc. 22nd Eur. Conf. Programm. Languages Syst., 2013 , pp. 574–592.
[14] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori and S. K. Rajamani, "Synergy: A new algorithm for property checking," in Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng., 2006, pp. 117–127
[15] J. W. Nimmer and M. D. Ernst, "Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java,"in Proc. 1st Workshop Runtime Verification, pp. 255–276, 2001
[16] K. Jamrozik, G. Fraser, N. Tillmann, and J. de Halleux, "Generating test suites with augmented dynamic symbolic Execution," in Proc. Tests Proofs, 2013, pp. 152–167.
[17] J. Malburg and G. Fraser, "Combining search-based and constraint- based testing," in Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng., 2011, pp. 436–439
[18] S. Ratcliff, D. R. White, and J. A. Clark, "Searching for invariants using genetic programming and mutation testing," in Proc. 13th Annu. Conf. Genetic Evolutionary Comput., 2011, pp. 1907–1914.
[19] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy dynamic symbolic execution for invariant inference," in ICSE. ACM, 2008, pp. 281–290.

.

.

All Rights Reserved © 2016 IJARBEST