Invariant Detection Using Enhanced Autoinfer

Sadia Ashraf, Almas Abbasi*
International Islamic University (IIUI), H10, Dept. of Computer Science, Islamabad, Pakistani.

* Corresponding author. Email: dralmas.iiurm@gmail.com
Manuscript submitted December 18, 2016, received May 15, 2017.
doi: 10.17706/jsw.12.10.816-823

Abstract: AutoInfer is a tool that is the state of the art in invariant detection. Invariants are properties of program components that remain unchanged throughout the execution of that component. AutoInfer automatically detects invariants for the programs under test which may or may not have a few invariant already present in them. AutoInfer uses AutoTest to generate a test suite for a given Program under test. The test suite (TS) is generated randomly using routine coverage as the coverage criteria. The generated TS is run on the program to create a change profile for the program, which in turn is used to activate relevant templates to generate quantified expressions. These expressions are the candidate contracts. Candidate contracts when run against the test cases are validated if they do not fail any test case. AutoInfer's results are heavily based on the generated Test Suite. The better the generated Test Suite is better the final contracts will be. The work in this paper proposes that using Whole Test Suite (WTS) generation instead of random generation to generate the TS will results in a faster generation of the TS and will capture more errors as compared to AutoInfer. WTS is the state of the art in TS generation so it will result in better coverage and an improved fault detection capability.

Key words: Invariant detection, test oracles, software testing, AutoTest.

1. Introduction

Software testing is the process in the life cycle of the software development that takes up the most time. The reason why so much time is taken in this phase is because in practice most of the testing is done manually. This emphasizes the need to automate the testing process. The literature contains a large body of work containing the test automation tools. These tools are aimed at saving time and cost that is incurred when testing is done manually [1].

Most of the test automation tools are TS generation [2] tools while a relatively smaller number attempt to generate automated oracles too. Oracles are files that contain the expected output of the programs under test. Oracles are used to validate if the test cases has produced the correct result or not [3]. Generally in practice the oracles are humans but there are tools that derive oracles automatically from formal and informal specifications. In the absence of the Specifications there are tools that derive oracles from other artifacts like program executions or documentation.

Invariant detection is one such technique where the programs behavior is captured in the form of invariants, post-conditions and pre-conditions. These three together form the contracts for a routine under consideration. Contracts are a form of oracle that can validate the subsequent versions of the program during regression testing [4].

AutoInfer is the tool that infers or detects invariants, post-conditions and pre-conditions for a program under test automatically. It basic working is based on the Daikon tool. The invariants are detected based on the templates that are selectively activated based on the change log for the Program under Test. First TS to exercise the Program under Test is generated randomly. This TS is used to create a change log for the programs routines. The routines are first instrumented, to capture the variable values after and before the routine is executed. This data is analyzed to detect possible invariants or contract for the routine. These expression capture the behavior of the program. All the candidate contracts are validated against the TS. The invariants that do not fail any test case are considered valid invariants.

The better the generated TS, the better the detected invariant will be. The more coverage they TS will achieve the better chance will the contracts have of capturing errors. It is therefore proposed to use Whole Test Suite Generation instead of proposed to use Whole Test Suite (WTS) [5] Generation instead of randomly generation to create the initial TS for AutoInfer. WTS is the state of the art that uses the genetic Algorithm (GA) to generate the entire TS evolutionally at the same time.

Section 2 contains the past work in the literature on invariant detection. Section 3 outlines the working of AutoInfer along with the possible improvements in the tool. Section 4 explains the proposed technique in detail and gives the pseudo code for the technique too. Section 5 contains the future work and section 6 contains the conclusion.

2. Related Work

Any system being developed needs to be tested to reduce the chances of failure. Early fault detection is the key to reducing the maintenance cost as well as the time and effort required to make a project succeed. Software testing is done throughout the software development life cycle to reduce cost and effort. It is said that almost 60% of the software development budget is used in testing.

Test cases are written to test different parts of the SUT (System Under Test) as the size of the system grows so does the Test Suite meant to exercise the system. Huge Test Suites increase computational cost as well as the human effort required to test the system. One solution where the human effort can be greatly reduced is by generating Test Suites automatically. The issue with automated Test Suite generation is that it generally produces prohibitively large Test Suites.

In testing an oracle is an entity that decides if a test case ran as expected or not. Generally the oracles in the development practices are humans; this adds unreliability and computational cost to the problem but if the oracles too are automated then no human intervention is required to complete the testing process.

Automated oracle generation can be divided into three parts based on the resources they are generated from [6]. Specified oracles are generated from formal specifications, derived oracles are generated from artifacts other than the specifications and implicit oracles are the oracles that mostly test quality traits instead of stated requirements.

All the past techniques that are used to automatically detect the invariants are considered in the literature. Major trends in context of the methods commonly employed to infer invariants are explored and the major limitation of the past work are also discussed

Invariants are properties, traits or qualities of the code that remain unchanged throughout the life of that piece of code. Example of an invariant can be the condition that a certain variable within a loop will never have a value above 10. Common invariants are loop invariant, class invariants etc. Invariants are added to the code in the form of assertions along with pre and post conditions. This is the main feature of contract based programming [7].

A technique by Earst et al. [8] that was implemented in a tool called Diakon [9] is a popular in automated invariant detection technique. The program under test is run using a chosen Test Suite and the likely invariants are detected at runtime dynamically.

Dynamic analyses for invariant detection can get computationally expensive therefor there have been techniques in literature that employ static analysis instead of dynamic analysis to infer invariants. These techniques are relatively less computationally taxing [10]-[13].

Another major trend observed is the techniques that make use of models to infer invariants; [14], [15] these techniques express the invariant in the form of models like FSM (Finite State Machines). Ratcliff et al. used mutation testing to aid in the detection of invariant through heuristics from search based software testing [16].

Wei et al's work [17], [18] attempts to improve the quality of the inferred invariants by generating the test suite in two part, the first set of test cases is used to infer invariant whereas the second set id designed to violate the inferred invariants thus making the test suite more capable of detecting faults.

3. AutoTest

Auto Test is used in the first step of AutoInfer [19]. It is used to generate the TS that is used to generate the change log. It uses random TS generation to generate TS' for the System Under Test (SUT). It first generates objects for the classes randomly by calling the constructors for the classes. These objects of different types are all stored in an object pool.

AutoTest is unit test generation tool i.e. it generates test cases for the each class. Each class is exercised by calling its routines using a randomly picked object from the object pool with randomly generated argument values. If a fitting object type is not present in the test Pool an object of that type is created from the class and added to the object pool as shown in Fig. 1 after the first step the randomly generated objects are then added to the object pool in the second step. If the object for a class does not exist in the pool the third step in Fig. 1 forms a loop where the missing objects for the classes are added to the object pool. In case the object of a class exists then that control is transferred to the third part of the Fig. 1 that runs the class routine.

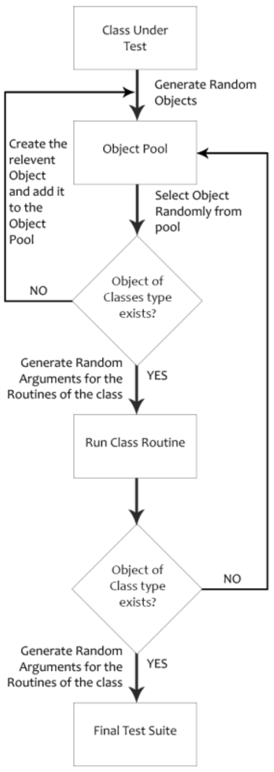
Class Under Test
Generate Random Objects
Object Pool
Create the relevent Object and add it to the Object Pool
Select Object Randomly from pool
NO
Object of Classes type exists?
Generate Random Arguments for the Routines of the class
YES
Run Class Routine
Object of Class type exists?
NO
Generate Random Arguments for the Routines of the class
YES
Final Test Suite

Fig. 1. Workflow of autotest.

The objects are randomly selected and run on the system until a predefined; user provided time frame is exceeded. The coverage criteria considered in AutoTest is the routine coverage.

Although this method for Test Suite generation is simple and straight forward, test generation using conventional methods is liable to get stuck in infeasible routines. Whole Test Suite generation method overcomes this problem by generating the Whole Test Suite at the same time evolutionally. Random Test generation may or may not cover all the areas of the system under Test or may take really long to reach a decent coverage. The resulting TS may have high redundancy resulting in wasted time and effort.

4. Proposed Model

Whole Test Suite Generation tool is the state of the art in test suite generation. A set of Test Suites that are randomly generated using random objects is created. All the Test Suites are put in a Test pool. The initial population is selected from the test pool. The number of Test Suites that are selected from the pool is taken as input from the tester.

The selected Test Suites are crossed over and mutated to produce the child generation. The Test Suites are ranked based on their fitness. The fitness of the test suite is based on how many routines it can cover. If a Test Suite that covers all the routines is generated then the algorithm terminates and gives that test suite as output but if no test suite is found that fulfill the fitness criteria then the selection, crossover and mutation operators are repeated until either the maximum number of iteration is reached or the fitness criteria is fulfilled..

5. Pseudo Code

5.1. Generate Initial Population

The first step is to generate the initial population for the genetic algorithm. The initial population consists of randomly generated test suites of varying sizes. Each Test Suite Consists of a random number of Test Cases while each Test Case is a set of random function calls. Like AutoTest the test suites are generated by randomly selecting objects.

```
ARRAY AllStatements[]: (Populate array with routine calls from the system under Test);

//Generate TestCases
FOR(i=0 TO MAXTestCase)
    ARRAY TestCase[] = RAND(AllStatements[])
END FOR

  //Generate TestSuites
FOR(i=0 TO MAXTestSuite)
    ARRAY TestSuite []= TestSuite[] + TestCase[RAND]
END FOR
```

5.2. Selection

The second step is to select the test cases based on their fitness to form the first parent population. The no of test Suites that are used for selection is taken as an input from the user and the population size can be changed in each run. AutoInfer does not contain selection crossover etc. because it generates test suites purely randomly. Selection is done based on the Roulette Wheel selection so that even the solutions with lower fitness have a chance and the diversity is maintained.

5.3. Crossover

Crossover is performed on the selected Test Suites by swapping the test cases in the test suites using One point Crossover. After crossover the child population of the same size as the parent population is generated. Elitism is ensured to preserve the solutions which have high fitness values. The selection therefor is done from the parent and the c child generations combined.

5.4. Mutation

Mutation in the Genetic Algorithm is meant to maintain diversity in the search space exploration. In WTS Gen the mutation is of three types. A random number is generated to decide what type mutation should be performed. The three possible types are i. interstion ii. Deletion and iii. Modification..

5.5. Main Flow of the Program

The Main flow of the program performs Crossover and Mutation on the selected Population in each iteration until the coverage criteria is fulfilled i.e. the fitness function f(T) becomes 0. The results from here are passed onto the next stage where the invariants are inferred.

```
// Perform Mutation on the child Population
MutationProb = RAND(1-3)
// Remove a Random Test Case
If MutationProb = 1
    MutationTestCase[] = TestCase[RAND]
    Remove MutationTestCase [RAND]

// Add a Random Test Case
ELSE IF MutationProb = 2
    MutationTestCase[] = TestCase[RAND]
    Add MutationTestCase [RAND]

// Replace a Random Test Case
ELSE IF MutationProb = e
    MutationTestCase[] = TestCase[RAND]
    Modify MutationTestCase [RAND]
END IF
```

```
Initialize Population()
```

```
      // Genetic Algorithm Iteration
   FOR(I=1 TO MAXIteration)
      Perform Selection()
      Perform Crossover()
      Perform Mutation()
      Check Fitness()
   IF (Fitness = RequiredFitness)
      Terminate program
       Output TestSuite
   END IF
   END FOR
```

6. Future Work

This hypothesis is still an idea under development. Experiments and empirical evaluation will allow us to gain more confidence in our hypothesis. Future work will contain empirically evaluating the idea and coding a tool for it is

7. Conclusion

This paper identified the limitations in the existing Autoinfer tool and proposed a method to overcome the limitation. It presents the hypothesis that Test Suite generation using WTS generation will yield better coverage in less time as compared to random test generation. The pseudo code for WTS generation is also presented. Diakon is the existing tool that is most commonly used for invariant detection, this concept papers proposes that daikon should be modified so that the part of the tool that generates test cases should do so intelligently based on 'Whole Test Suite' generation. The direction of the future work is to implement the proposed concept and perform empirical studies to verify or disprove the claims made here.

References

[1] Enoiu, Eduard P., *et al*. (2016). A comparative study of manual and automated testing in industrial embedded software. *Proceedings of The International Conference on Testing Software and Systems* ICTSS.

[2] Matinnejad, R., *et al*. (2016). Automated test suite generation for time-continuous simulink models. *Proceedings of the 38th International Conference on Software Engineering.*

[3] Jahangirova, G., *et al*. (2016). Test oracle assessment and improvement. *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

[4] Wang, B., Hongbiao G., & Jingde, C. (2016). Contract-based programming for future computing with Ada 2012. *Proceedings of the 2016 International Conference on Advanced Cloud and Big Data*.

[5] Rojas, José M., *et al.* (2016). A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*.

[6] Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2013). A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*.

[7] Wächter, H., & Reuter, A. (1991). The contract model. *Datenbank Rundbrief*, *8*, 83-85.

[8] Ernst, M. D., *et al*. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*.

[9] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., & Xiao, C. (2007). The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, *69(1)*, 35-45.

[10] Beyer, D., Henzinger, T. A., Jhala, R., & Majumdar, R. (2005, April). Checking memory safety with Blast. *Proceedings of the International Conference on Fundamental Approaches to Software Engineering* (pp. 2-18).

[11] Tillmann, N., Chen, F., & Schulte, W. (2006, November). Discovering likely method specifications. *Proceedings of the International Conference on Formal Engineering Methods* (pp. 717-736).

[12] Flanagan, C., & Leino, K. R. M. (2001, March). Houdini, an annotation assistant for ESC/Java. *Proceedings of the International Symposium of Formal Methods Europe* (pp. 500-517).

[13] Singh, R., Giannakopoulou, D., & Păsăreanu, C. (2010, July). Learning component interfaces with may and must abstractions. *Proceedings of the International Conference on Computer Aided Verification* (pp. 527-542).

[14] Heule, M. J., & Verwer, S. (2013). Software model synthesis using satisfiability solvers. *Empirical Software Engineering, 18(4)*, 825-856.

[15] Walkinshaw, N., Bogdanov, K., Derrick, J., & Paris, J. (2010, November). Increasing functional coverage by inductive testing: a case study. *Proceedings of the IFIP International Conference on Testing Software and Systems* (pp. 126-141).

[16] Ratcliff, S., White, D. R., & Clark, J. A. (2011, July). Searching for invariants using genetic programming and mutation testing. *Proceedings of the 13th annual conference on Genetic and evolutionary computation* (pp. 1907-1914).

[17] Polikarpova, N., Ciupa, I., & Meyer, B. (2009, July). A comparative study of programmer-written and automatically inferred contracts. *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (pp. 93-104). ACM.

[18] Wei, Y., Furia, C. A., Kazmin, N., & Meyer, B. (2011, May). Inferring better contracts. *Proceedings of the 33rd* International *Conference on Software Engineering* (pp. 191-200).

[19] Wei, Y., Roth, H., Furia, C. A., Pei, Y., Horton, A., Steindorfer, M., & Meyer, B. (2011, November). Stateful testing: Finding more errors in code and contracts. *Proceedings of 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE),* (pp. 440-443). IEEE.

Sadia Ashraf was born in Dubai on 3rd March 1984. She completed her high school in Dubai and her major subjects were math, physics and computer science. After completing high school. She shifted to Pakistan where she did her BSand MS in software engineering from International Islamic University Islamabad (IIUI). She is currently a phd candidate at the same University.

She has an experience of 4 years, working as front end developer in different software houses. During her time in software houses she work on Government level projects as well as with international client from all over the globe. Her clients were from, Australia, U.S, Europe and parts of Asia. She is currently serving as a lecturar in the National University of Modern Languages (NUML).