M. Tech. (Computer Science) Dissertation Series

Clustering and Ranking unit bugs using dynamic invariants

a dissertation submitted in partial fulfillment of the requirement for the M. Tech. (Computer Science) degree of the Indian Statistical Institute

By

Nehul Jain

under the supervision of

Dr. Ansuman Banerjee

INDIAN STATISTICAL INSTITUTE
203, Barackpore Trunk Road
Calcutta - 700 108

To my dear parents ... for standing by me,
my supervisor ... for his constant support

Acknowledgements

My M.Tech duration has been a journey surrounded by wonderful people. This thesis would not be complete without acknowledging them who were with me through this journey.

To begin with, I wish to thank my parents for allowing me to spend these two long years, away from them, and my sister and her family for providing me the home I needed so much. Their presence and encouraging words kept me moving.

I fall short of words to acknowledge my deep sense of gratitude towards my supervisor Dr. Ansuman Banerjee for his invaluable guidance, motivation and constant encouragement throughout the course of this work. I feel indebted to him for giving me the best problems and relying on my abilities to solve them.

I would like to thank the faculties of the institute for their support during my M.Tech. tenure. In particular, I would like to thank Prof. Susmita Sur-Kolay for her encouragement and support.

Last but not the least, I would take this opportunity to thank all my friends. In particular, I would like to take this opportunity to thank Saikat Dutta from Jadavpur University for his tremendous help and contribution in this work.

Nehul Jain
Indian Statistical Institute, India

Abstract

Unit testing and verification constitutes an important step in the validation life cycle of large and complex multi-function software code bases. Many unit verification methods often suffer from the problem of false negatives, when they analyse a function in isolation and look for errors. It often turns out that some of the reported unit failures are *infeasible*, i.e. the valuations of the function input parameters that trigger the failure scenarios, though feasible on the function in isolation, cannot occur in practice considering the integrated code, in which the function-under-test is instantiated. To address this problem, we present in this paper, an automated two-stage failure scenario classification and prioritization strategy that can filter out false negatives and cluster them accordingly. In the first stage, we use Daikon to identify input constraints on the function under test, by mining invariants on the function boundary. Each invariant thus generated is assigned a confidence by Daikon. In the second stage, we associate a confidence to each failure scenario by analysing the invariants they are in conflict with, and compute their respective confidence values. Failures with similar characteristics are clustered together and assigned the same confidence value. The confidence values enable us assign priorities to the failure scenarios and help us prioritize the bug fixing activity. Experiments show that our clustering and prioritization scheme works quite well in practice.

Contents

1 Introduction **4**

1.1 Motivation and Objective 9

1.1.1 A toy example 12

1.2 Summary of Contributions 16

2 Background and Related Work **19**

2.1 Static analysis approaches 19

2.1.1 CEGAR 21

2.2 Dynamic program analysis 22

2.3 Invariants 23

2.4 Daikon 24

2.5 Test Case Prioritization 25

3 Detailed Methodology **26**

3.1 Illustration on the toy example 29

3.1.1 Invariant Mining with beliefs 31

3.1.2 Failure scenario Ranking 32

3.2 Flow of Detailed Methodology 34

3.2.1 Invariant Miner 34

3.2.2 Confidence 35

3.2.3 Constraint analyzer 36

3.2.4 Scenario Ranker 36

4 Applications 38
4.1 Clustering and prioritization 38
4.2 Localization 41
4.2.1 Our Method 43

5 Implementation 45
5.1 System Architecture 45
5.2 Failure Extraction 45
5.3 Invariant Mining 46
5.4 Confidence Measure 48
5.5 Clustering of Failure scenarios 48
5.6 Ranking 49

6 Evaluation 50
6.1 Experience with replace 50
6.1.1 Testing with CREST 51
6.1.2 Invariant Mining 52
6.1.3 Confidence Measure and filtering 54
6.1.4 Failure scenario Clustering and Ranking 55

7 Conclusion and Future Work 56

List of Figures

1.1 *Infeasible failure scenario* 6
1.2 *Framework* 10

3.1 *Process Outline* 28
3.2 *Invariants on Simple example* 32
3.3 *Framework* 35

4.1 *Framework Application* 39
4.2 *Invariants on Application example* 43

Chapter 1

Introduction

Verification is a significant chunk of a design cycle that is lengthening. Bugs are frequent on software. Software bugs are expensive. Failures at unexpected time can cost lives and money. As society becomes increasingly software-dependent, bugs also reduce our productivity and threaten our safety and security. Decreasing these direct and indirect costs is desirable as well as a necessity.

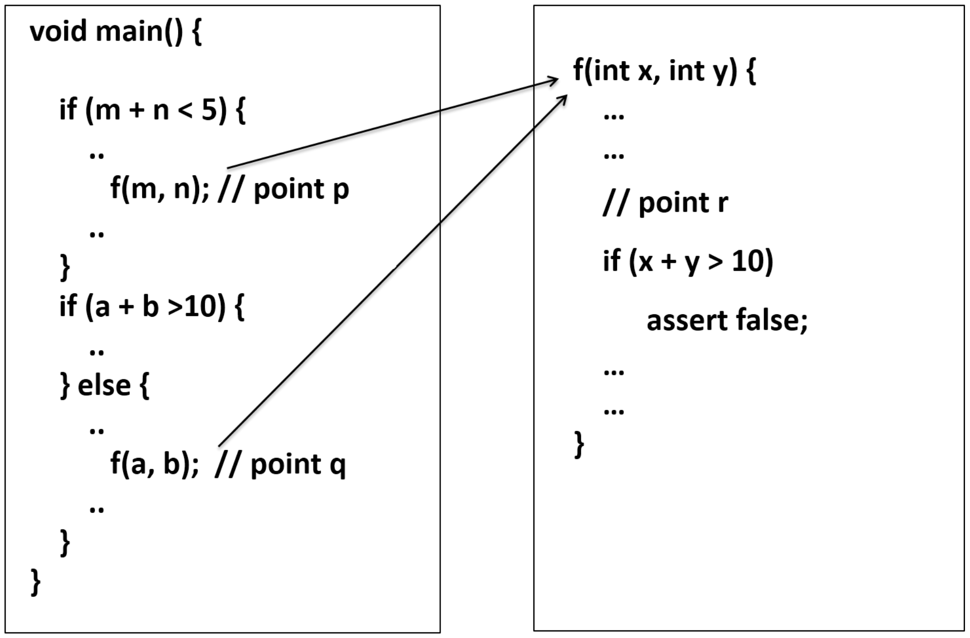
Even though augmented with extensive documentation, software often behaves in completely surprising ways [11]. The complexity of verification lies in the complexity of the software itself. Documentations are expected to include all possible details for ease of understanding and maintenance. Important information is often hidden in a mass of irrelevant detail making bugs hard to detect. Design mistakes are often discovered too late, making it expensive or even impossible to correct them.

There are two basic approaches to validation- testing and formal verification. Software testing is done by running the executable (obtained by compilation) on multiple inputs usually on the target platform. Testing is a widespread validation approach in the software industry and has a lot of benefits. Testing can be (partially) automated. It has been seen that it can detect a lot of bugs. Formal verification proves or disproves the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics, hence leaving no doubt on the correctness of verified software. Formal methods can reveal errors that may not be detectable by million cycles of testing. This aspect is especially valuable in security assurance, because security attacks often exercise an application in unforeseen and untested ways.

A major drawback of testing on critical systems is that it is very costly and time consuming. Testing is not exhaustive and can leave hidden errors. These errors may show up later at unexpected times after deployment which is highly undesirable. Model checking also has its own challenges. Complex data types, pointers, finding good abstractions, generating complex invariants are some of them which are not completely resolved. Checking functional properties, exploiting modularity, and achieving scale with respect to data and concurrency are issues that need to be addressed for verifying large softwares.

Professional coding practices typically advocate the development of a large complex software code base as a collection of functions and their interaction. Each function typically consists of a set of parameters, appropriate valuations of which determine the context in which the function will be instantiated. The execution of the entire software is the organized orchestration of the control and data flow induced by the top level code, with inline function calls in between to fulfil the top level design objective. Such modular design styles not only facilitate development but also diagnosis and debug.

Verifying correctness of a software code at a large scale has always been a grand challenge. Traditional test methods typically run out of steam, considering the fact that the number of test scenarios arising out of the possible orchestrations of the different functions and their instantiations, is beyond the limit of what they can achieve in reasonable time. Formal verification methods, on the other hand, attempt at exhaustive verification of abstractions of the underlying infinite software state space, with limits on the amount of promise they can deliver. The complex state space arising out of the possible interleaving and instantiations, typically give rise to an



```
void main() {

  if (m + n < 5) {
    ..
      f(m, n); // point p
    ..
  }
  if (a + b >10) {
    ..
  } else {
    ..
      f(a, b);  // point q
    ..
  }
}
```

```
f(int x, int y) {
  ...
  ...
  // point r
  if (x + y > 10)
      assert false;
  ...
  ...
}
```

Figure 1.1: *Infeasible failure scenario*

enormous state space, traversal of which is an infeasible proposition.

A popular practical approach often found to be successful in practice is modular testing or verification. A modular approach essentially treats each module in isolation and tries to come up with an exhaustive guarantee on its functional correctness. Unit testers typically target some coverage criterion and generate test scenarios to achieve a reasonable proportion of them in as much time as they can. A number of unit bugs are expected to be revealed as an outcome of this exercise, leading to possible refinements of the buggy modules. Formal approaches for modular analysis essentially attempt to analyse (either symbolically or explicitly) every possibility of the presence of a bug inside the unit, and attempt to prove their presence, usually guided by an assertion violation or reachability of error labels. This can possibly lead to some quick unit level violations, which can be diagnosed and fixed.

While modular approaches are quick and scalable in finding unit bugs, they often suffer from the problem of false negatives. A failure scenario reported by a unit tester may actually be infeasible, that is, the test data that triggered the failure scenarios, although reasonable when looking at the unit-under-test in isolation, are outside of their boundaries when considering the integrated software code. Similarly, failure scenarios received from a formal verifier analysing a unit in isolation, may actually be spurious, considering the fact that the state pointed to may be unreachable in the integrated code base, or the failure run may not actually occur. In both the cases, these false negatives received from a unit level verifier, need to be diagnosed and analysed for correctness. False alarms may lead to needless fixes, which has to be avoided. Figure 1.1 shows an infeasible failure scenario. The function f(int x, int y) may report failure scenarios for unit tests which drive input values such that the condition $(x + y > 10)$ is met at point r. However, when put in the context of $main$, it is called at program points p and q, where the input sets (m, n) and (a, b) have their value constraints. It is clearly evident that due to the guard conditions at the points from which f is called, the error triggering condition can actually never happen in practice. Therefore, this failure scenario is spurious and needs to be filtered out.

To get around this problem, researchers have typically worked on enriching the unit level verifier with more knowledge about the environment. It is acknowledged that an exhaustive scalable verification solution at the integrated system level is not easy to achieve, and unit validation is a much needed step before attempting to scale up to the system level. Researchers have suggested enriching the unit verifier with as much knowledge about the unit's instantiating environment as possible to get around the false negatives. On one extreme, several researchers have looked at the possibility of unit verification, in universal environments, assuming all valuations and combinations of the unit parameters in all scenarios. While this leads to better theoretical guarantees, this is not a viable option, considering the complexity of the underlying verification problems. On the other extreme, several articles report the possibility of refinement style of reasoning, which may start with zero knowledge about the instantiating environment, and incrementally add as much revealed in the false verification steps. None of these approaches have been reported to be successful at the large scale.

A viable alternative to get around the problem of false negatives, is to possibly analyse and rule out the scenarios which exhibit call sequences / valuations, which will never occur in the integrated code. While it is possible for the developer to actually achieve this in practice, the number of such scenarios may be overwhelming and end up in a needlessly painstaking exercise. Moreover, in a distributed development environment, and a concurrent code base, it may actually be an infeasible proposition to achieve this plan.

The motivation of our work is as follows. Each reported failure scenario can be best analysed from the perspective of the information about the calling environment that the scenario assumes. Failures which depict scenarios that are in more conformance to the calling environment, should be examined with more priority, since they possibly depict true bugs. The ones, which assume calling valuations that contradict common knowledge about the function's environment, have a greater chance to be spurious. Moreover, many of the failures may actually relate to similar flows in the code, and need not be separately examined. Intuitively, the infeasible failure scenarios tend to cluster because they are relatively likely to relate to one specific violated input constraint. Inspecting one failure scenario within the cluster yields information about the others. For example, if the failure scenarios in one cluster are perfectly correlated, then classifying one effectively classifies them all. If one failure scenario is identified as infeasible, so are the others and can be skipped. If one failure scenario relates to a true bug, so may be the others and need to be examined.

We present in this work, an automated framework for failure scenario classification and prioritization. Our proposed methodology has two steps as explained below.

- In the first stage, we use Daikon [7] to identify input constraints on the function under test, by mining invariants on the function boundary. Each invariant thus generated is assigned a confidence by Daikon.

- In the second stage, we associate a confidence to each failure scenario by analysing the invariants they are in conflict with, and their respective confidence values.

Failures with similar characteristics are clustered together and assigned the same confidence value. The idea behind this clustering is that, failures conflicting with the same invariants might be manifestations of similar errors. The confidence values enable us assign priorities to the failure scenarios and help us prioritize the bug fixing activity. This work is inspired by a similar work on ranking and classifying unit level failure scenarios for hardware logic code bugs, in [13].

We performed an empirical evaluation of this framework on replace, a program written in C which is a part of Siemens Benchmark Suite [10]. Results demonstrate that our proposed method works quite well in practice.

1.1 Motivation and Objective

The problem in this thesis can be articulated with the help of Figure 1.1. **S** represents the software in which the module to be tested resides. **f** is the function which we want to verify.

It would be easy to verify function f in isolation. But, this approach often suffers from the problem of false negatives. A failure scenario reported by a unit tester may actually be infeasible, that is, the test data that triggered the failure scenarios, although reasonable when looking at f in isolation, are outside of their boundaries when considering the context of $main$. Similarly, failure scenarios received by formally verifying or analysing a unit in isolation, may actually be spurious, considering the fact that the state pointed to may be unreachable in the context. In both the cases, these

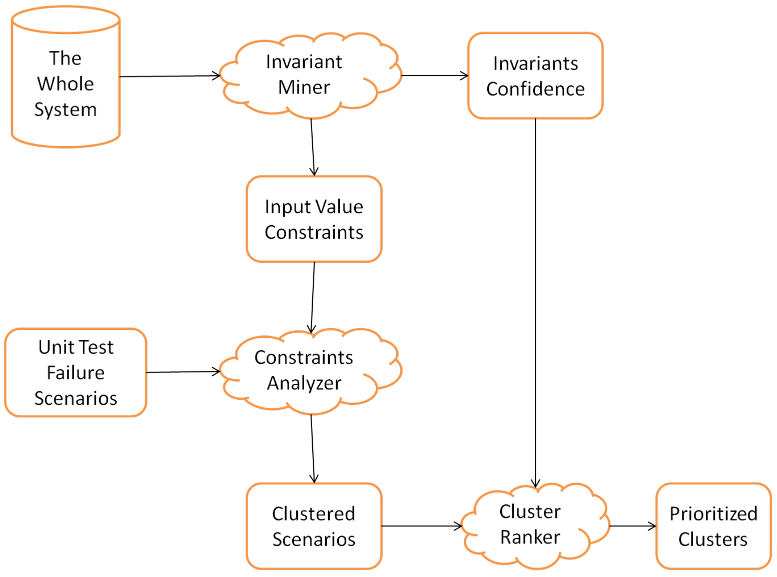


Figure 1.2: *Framework*

false negatives received from a unit level verifier, need to be diagnosed and analysed for correctness. False alarms may lead to needless fixes, which has to be avoided.

The main challenge in this problem is to determine which scenario can be created by the context, and verify whether a given unit level failure scenario can at all be simulated in the software. In this example we would like to know can f at all be called when condition $(x + y > 10)$ holds. A viable alternative to get around the problem of false negatives, is to manually analyse and rule out the scenarios which exhibit call sequences / valuations, which will never occur in the integrated code.

While it is possible for the developer to actually achieve this in practice, the number of such scenarios may be overwhelming and end up in a needlessly painstaking exercise. Moreover, in a distributed development environment, and a concurrent code base, it may actually be an infeasible proposition to achieve this plan. If we try to solve the problem using static analysis of program to derive the feasibility or infeasibility of a scenario we are faced with scalability issues of the method. The motivation of this problem is to reduce the manual effort spent in ruling out infeasible scenarios.

The objective of this thesis is to use dynamic invariants to aid the existing approaches to verifying a function. Instead of using formal verification or test suite as a singular technology for verification problem described above, this thesis aims to use bug scenarios on f rather than on S to improve coverage of relevant behaviour of f and restricts the attention of the verification engineer to those bug scenarios on f that are more likely to occur in S. Specifically this thesis addresses the following broad objectives.

1. Applications are usually built by assembling various robust reusable software components together. Within a system S, a robust code module f written for general use can be used. It may be later customized for specific purpose by making small changes. If we check f in isolation a lot of errors may pop up due to the customization of our module which takes care of only those cases of inputs possible in our scenario. These scenarios may be limited by the system S as input to f. These spurious errors need to be ignored over errors that may be introduced due to the removal/addition of control flow paths that are required/not-required. It is not feasible for a verification engineer to examine all failure scenarios and determine their correctness. The objective of this work is to extract knowledge of the environment of f in the form of dynamic invariants and use these to filter out failure scenarios in terms of their likelihood of being real.

2. Software debugging is done in various contexts. Software evolution is an ongoing process. Our methodology can be useful in program evolution. Usually components are modified or cleaned for efficiency, portability or readability. In such a case there might exist a very bug prone module f in a system S. Whenever any minor

changes are made to the system, it is required that the robustness of f be rechecked. We would like to see if the changes in S can lead to a scenario where function f does not work fine.

3. Given a set of failure scenarios on a function f, when ranking failure scenarios on their occurrence in real scenarios, we would also like to minimize the scenarios to be actually examined by the developer manually. We would not like a developer to examine the manifestation of a single error more than ones. We would like to save such an effort and try to give error scenarios which point out more bugs in less number of examinations done by a developer.

1.1.1 A toy example

The problem in this thesis can be illustrated with the help of a very simple example listed below. Program S represents the software in which the module to be tested resides. int div_by_f (int c, int f) is the function which we want to verify. Bug scenarios on int div_by_f (int c, int f) are given, derived either through unit testing or formal verification of int div_by_f (int c, int f).. It is expected to compute some non-zero function of c and f and return a value. When return value is 0, it can be considered as an erroneous return where the function does not adhere to the requirement.

```
//Program S
//Targeted module div_by_f( int c, int f)
  int gcd(int u, int v) {
    int t;
      if(v == 0) v = u;
      while (v) {
        t = u;
        u = v;
        v = t % v;
      }
     u = u < 0 ? -u : u; /* abs(u) */

     return u == 0 ? 1: u;
 }
```

```c

int div_by_f( int c, int f){
    int i=0,result;

    if(c >0){
         i = c + 20*f;
         printf(”%d”,i);
     }
     if(i%f != 0)
         return 0;//error
     if(f== 0)
         return 0;//error
     result = i/f;
     if(result < 0)
         return 0;//error

   return result;
}

int main(int argc ,char* argv[]){

  int s,t,a,v;

    scanf(”%d %d %d”,&s,&t,&a);

    v = gcd(s,t);

    x = div_by_f(s, v);
    assert(x);

    switch(a)
    {
        case 1:
        x = div_by_f(t, v);
        break;

        case 2:
        x = div_by_f(s, v);
        break;

        case 3:
        x = div_by_f(s+t, v);
        break;

        case 4:
        x = div_by_f(s*t, v*v);
        break;
    }

    assert(x);
  return 0;
}
```

Simple Example

Unit testing isolates each part of the program and shows that the individual parts are correct. If we verify the function div_by_f() in isolation, a unit tester may give a lot of error scenarios. For example, it can give scenarios corresponding to the following evaluations when a call to div_by_f() is made

```
unit-under-test:div_by_f()
1. (c=11,f=2)
2. (c=-189,f=9)
3. (c=10,f=0)
```

As a unit tester has no knowledge of the conditions under which a call to div_by_f() can be made, it might give a lot of scenarios which are not feasible at all. For example, scenario 1 is never possible in the current context of the function being called as the environment forces the condition $c\%f == 0$ as f is always a factor of c. Scenario 2 on the other hand, is possible in the given context of the function. For evaluation $s = -189, t = 153, a = 2$, f is called with given parameters $c = -189, f = 9$ in the failure scenario and hence this failure is real. Also scenario 3 is never possible in the current context of the function being called as the environment also forces the condition $f >= 1$ as f is always, a gcd of two numbers or product of two gcds, across all calls.

Thus, when presenting these scenarios to a developer for debugging, we would like to remove scenarios which are not possible. Our objective is to classify spurious and real errors among all the error scenarios presented by testing a unit-under-test.

A natural question that arises is ”How do we at all know the kind of conditions imposed by the environment?” Until and unless these conditions are specified by the developer we may not know them. Even if they are specified in a documentation, constant updation to a software may result in violation of such conditions. ”How do we know the most current situation?” To get around this problem, researchers have typically worked on enriching the unit level verification problem with more knowledge about the environment. It is acknowledged that the verification problem at the integrated system level is not easy to achieve, and hence, researchers have suggested enriching the unit verified with as much knowledge about the unit's instantiating environment as possible to get around the false negatives. On one extreme, several researchers have looked at the possibility of unit verification, in universal environments, assuming all valuations and combinations of the unit parameters in all scenarios. While this leads to better theoretical guarantees, this is not a viable option, considering the complexity of the underlying verification problems. On the other extreme, several articles report the possibility of assume guarantee style of reasoning, which may start with zero knowledge about the instantiating environment, and incrementally add as much revealed in the false verification steps. None of these approaches have been reported to be successful at the large scale. In this thesis we try to answer these questions with the help of dynamic invariants.

How will formal verification work? Formal verification proves or disproves the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods. For small programs it is doable but for large software it is not feasible. In reality programs are run on a computer with bounded memory. But even with bounded memory,complexity of verification in practice is too high for a finite-state model-checker to verify within a reasonable amount of time.

That is formal methods do not scale to our requirements.

Will static invariants serve our purpose? Static invariants are derived using the analysis performed without actually executing programs. Analysis is performed directly on the source code. We could use static invariants to rule out certain failure scenarios. A failure scenario contradicting even a single invariant is definitely a spurious failure and hence can be ruled out completely, and need not be considered at all by the developer and the need of ranking does not come into picture. Our methodology uses invariants to determine whether a failure scenario is feasible or not. It does not seem to restrict its use to dynamic invariants. It definitely allows the use of static invariants in its framework. But static invariants have their own limitations. The issues that arise in static invariants is that, scalability of existing approaches to static invariant generation is severely limited due to the high computation cost of the underlying symbolic reasoning techniques [8] and cannot mine invariants from complex data structure representations. These limitations encourage the use of dynamic invariants which do not suffer from scalability issues. A failure scenario that contradicts an invariant with high confidence is likely to be false.

1.2 Summary of Contributions

Concretely our objective can be stated as below.

Problem We are given $< S, f, TC >$ consisting of

- S is the software in which the targeted module lies.
- f is the targeted module which belongs to the software S.

- TC is the set of bug scenarios or failure scenarios on module f. TC contains n number of bug scenarios.Each such scenario gives a faulty execution of f.

We would like to-

- Classify and rank the bug scenarios among the test cases to be presented to the developer for correction in decreasing order of confidence, rank or belief in the scenario.

We study the problem of ranking failure scenarios on a module of a software. Concretely, the contributions of this work are as follows-

1. Our ranking method focuses on a specified module f which is to be targeted and ranks failure scenarios so that higher ranked failure scenarios are more likely to be true in the given software. This method combines dynamic invariants mined on the software with buggy test scenarios for the original module.

2. We also classify failure scenarios into different groups depending on the type of environment they provide to the module being tested.

So we present a ranking and clustering methodology based on the dynamic behavior of the system at certain program points. This behaviour is characterised by invariants, that hold at a point and, their confidence measures.

Organisation The thesis is organised as follows-

Chapter 1 This is the introductory chapter which discusses the objective and motivation of the problem addressed in the dissertation.

Chapter 2 This chapter presents background literature in the area of this research.

Chapter 3 This chapter presents a detailed methodology for invariant guided bug prioritization.

Chapter 4 This chapter presents certain applications and possible future work regarding invariant guided bug prioritization.

Chapter 5 This chapter give details of the various tools used to implement our framework.

Chapter 6 This chapter presents a case study on replace which is a part of Siemens Benchmark Suite.

Chapter 7 This chapter summarizes our contributions and scope for future work.

Chapter 2

Background and Related Work

The primary aim of this chapter is to provide some background concepts that are necessary for developing the foundations of the thesis. The goal of verification in general is to assure that software fully satisfies all the expected requirements. There are two fundamental approaches to verification. First is dynamic verification, also known as Test or Experimentation. This is good for finding bugs. In simple words testing is executing a system in order to identify any gaps, errors. Second is static verification, also known as Analysis.

2.1 Static analysis approaches

Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool. Static analysis can be applied to formally verify properties of a given program.

Static analysis, which automates the abstraction of the program execution, always terminates. Some of the implementation techniques of formal static analysis include Model checking, Data-flow analysis, Abstract interpretation, Assertions, Symbolic execution.

Model checking considers systems that have finite state or may be reduced to finite state by abstraction [3]. Specifications are expressed in temporal logic, and the system is modeled as a state transition graph. An efficient search procedure is used to determine whether or not the state transition graph satisfies the specifications.

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program [4]. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program.

Abstract interpretation models the effect that every statement has on the state of an abstract machine (i.e., it 'executes' the software based on the mathematical properties of each statement and declaration) [5]. This abstract machine over-approximates the behaviours of the system: the abstract system is thus made simpler to analyze, at the expense of incompleteness. Not every property true of the original system is true of the abstract system. If properly done, abstract interpretation is sound. In other words, every property true of the abstract system can be mapped to a true property of the original system.

Use of assertions in program code was first suggested by Hoare logic [9]. There is tool support for some programming languages e.g., the SPARK programming language which is a subset of Ada and, the Java Modeling Language JML using ESC/Java and ESC/Java2, Frama-c WP (weakest precondition) plugin for the C language extended with ACSL (ANSI/ISO C Specification Language).

Symbolic reference is used to derive mathematical expressions representing the value of mutated variables at particular points in the code [12]. Instead of executing a program on a set of sample inputs, a program is ”symbolically” executed for a set of classes of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases.

2.1.1 CEGAR

Many approaches have been considered for formal verification, all of them being approximations of the program semantics (formally defining the possible executions in all possible environments) formalized by abstract interpretation theory.

Deductive methods produce formal mathematical correctness proofs using theorem provers or proof assistants and need human interaction to provide inductive arguments (which hardly scales up for large programs which are modified over long periods of times) and help in proofs (such as proof hints or strategies);

Model checking exhaustively explores finitary models of program executions, which can be subject to combinatorial explosion, requires the human production of models (or may not terminate in case of automatic refinement of the model). An alternative is to explore partially the model but this is then debugging, not verification.

Static analysis techniques which automates the abstraction of the program execution, always terminates but can be subject to false alarms (that is warnings that the specification may not be satisfied although no actual execution of the program can violate this specification)

The state explosion problem remains a major hurdle in applying model checking to large industrial designs [2]. Abstraction is certainly the most important technique for handling this problem.

This technique computes an upper approximation of the original program. Thus, when a specification is true in the abstract model, it will also be true in the concrete design. However, if the specification is false in the abstract model, the counterexample may be the result of some behavior in the approximation which is not present in the original model. When this happens, it is necessary to refine the abstraction so that the behavior which caused the erroneous counterexample is eliminated. In CEGAR information from the erroneous counterexample is used to refine the abstraction. This technique is useful in keeping the size of the abstraction small and hence manageable.

2.2 Dynamic program analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior. Use of software testing techniques such as code coverage helps ensure that an adequate slice of the program's set of possible behaviors has been observed. Also, care must be taken to minimize the effect that instrumentation has on the execution (including temporal properties) of the target program. Inadequate testing can lead to catastrophic failures

There has been a lot of work on techniques combining static and dynamic analysis. A good example would be a tool named DART, for automatically testing software. DART (Directed Automated Random Testing) takes the best of both approaches (precision of dynamic analysis AND efficiency of static analysis) It combines three main techniques:

1. automated extraction of the interface of a program with its external environment using static source-code parsing;

2. automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in; and

3. dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths.

Together, these three techniques constitute Directed Automated Random Testing, or DART for short. The main strength of DART is thus that testing can be performed completely automatically on any program that compiles there is no need to write any test driver or harness code. During testing, DART detects standard errors such as program crashes, assertion violations, and non-termination.

2.3 Invariants

A program invariant is a property that is true at a particular program point or points, such as might be found in an assert statement, a formal specification, or a representation invariant. Examples include $y = 4 * x + 3$; $x > abs(y)$; $array\, a\, contains\, no\, duplicates$; $n = n : child : parent(for\, all\, nodes\, n)$; $size(keys) = size(contents)$;. Invariants explicate data structures and algorithms and are helpful for programming tasks from design to maintenance. Despite their advantages, invariants are usually missing from programs. An alternative to expecting programmers to fully annotate code with invariants is to automatically infer likely invariants from the program itself. Invariants are valuable in many aspects of program development, including design, coding, verification, testing, optimization, and maintenance. They also enhance programmers' understanding of data structures, algorithms, and program operation. Invariants can be used in numerous ways. They can be used for documentation as they characterize certain aspects of program execution, and refine documentation as automatically determined invariants can give the most recent information. Assumptions and assertions can be cross checked with determined invariants. When a programmer has knowledge of invariants, bugs inserted due to their violation can be avoided.

Invariant detection recovers a hidden part of the design space: the invariants that the programmer had in mind. This can be done either statically or dynamically. Static analysis examines the program text and reasons over the possible executions and runtime states. The most common static analysis is dataflow analysis, with abstract interpretation as its theoretical underpinning. The results of a conservative, sound analysis are guaranteed to be true for all possible executions. Static analysis has a number of limitations. It cannot report true but undecidable properties or properties of the program context. Static analysis of programs using language features such as pointers remains beyond the state of the art because the difficulty of representing the heap forces precision-losing approximations and produces weak results. Dynamic analysis, which runs the program, examines the executions, and reports properties over those executions, does not suffer these drawbacks and so complements static analysis

2.4 Daikon

Daikon is an implementation of dynamic detection of likely invariants [7]; that is, the Daikon invariant detector reports likely program invariants.

Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data. Daikon can detect invariants in C, C++, Java, and Perl programs, and in record-structured data sources.

Dynamic analysis, runs the program, examines the executions, and reports properties over those executions. Dynamic invariant detection occurs in three steps: the program is instrumented to write data trace files, the instrumented program is run over a test suite, and then the invariant detector reads the data traces, generates potential invariants, and checks them, reporting appropriate ones.

2.5 Test Case Prioritization

Test case prioritization techniques schedule test cases in an execution order according to some criterion. The purpose of prioritization is to increase the likelihood that if the test cases are used for regression testing in the given order, they will more closely meet some objective than they would if they were executed in some other order. Test case prioritization can address a wide variety of objectives, including increasing the rate of fault detection, the rate of detection of high-risk faults, the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process, their coverage of coverable code in the system, their confidence in the reliability of the system under test at a faster rate. In practice, and depending upon the choice of objective, the test case prioritization problem may be intractable: for certain objectives, an efficient solution to the problem would provide an efficient solution to the knapsack problem. Thus, test case prioritization techniques are typically heuristics with the goal to optimize rate of fault detection. Various heuristics have already been explored for a specific objective function [14]. Some of them are randomized ordering, prioritizing in order of coverage of branches, prioritizing in order of coverage of branches not yet covered, prioritizing in order of total probability of exposing faults, prioritizing in order of total probability of exposing faults adjusted to consider effects of previous tests prioritizing in order of coverage of statements, prioritizing in order of coverage of statements not yet covered. Many more have been suggested in [6].

We have introduced some work on techniques combining static and dynamic analysis. In this work, we present a technique that combines unit testing or model checking of a functional unit with use of dynamic invariants. A goal of this work is to suggest, a means to rank and cluster a bunch of failure scenarios on a function that need to be examined by developers.

Chapter 3

Detailed Methodology

In this section, we elaborate on the different steps of our method. Large software consist of smaller functional units. These functional units interact among themselves by getting inputs and providing outputs among themselves. Although verification of these huge software is not scalable it can be robust for smaller functional units. Our work gives a method to have some advantage of verification of smaller functional unit, in its verification in a huge software, without compromising hugely on the robustness.

When verifying a module it may either be done by testing or formally verifying the module. While testing a module it may be modeled as a closed system by assuming inputs from the environment as being generated non deterministically by the system itself. Testing a component in a free environment gives a good coverage for the individual component although it might not be exhaustive. Formal verification on the other hand will be exhaustive and give all possible errors if the technique is applicable to the targeted module.

A function being verified may be called from different functions with widely different parameters. When verifying the function the conditions imposed by these different functions may not be known. This may result in hundreds of failure scenarios on the same function which might be too many to be examined by a programmer. Many of these failure scenarios may be fictitious. That is, the test data that triggered the failure scenarios, although reasonable when looking at the function in isolation, is actually outside of its input boundaries when considering the whole system. In other words, the unit validation process may have generated many infeasible scenarios. Bugs may be cases where the given module cannot handle a certain kind of environment which is feasible in the software. We want to help catch this unhandled environment as early as possible. So we need a way to rank the failure scenarios. We rank them on the confidence of occurrence of the certain kind of environment in the software.

Also if one failure scenario from this unhandled environment is taken care of, it might lead to elimination of other failure scenarios arising in the same environment. Thus bugs providing a similar kind of environment to a module are grouped into a single bug family. Analysing one will take care of the rest. Thus clustering might reduce the number of failure scenarios to be examined by the programmer.

Our objective is to rank failure scenarios produced on a module such that higher ranked ones are more likely to be real. We present the failure scenarios to the designer in decreasing order of ranks.

Contribution of this work is a ranking and clustering methodology based on dynamic invariants and confidence measures. We rank them in a way such that higher ranked ones more likely to be real, with the aim to facilitate faster recognition of true errors, and reducing some manual effort.

This thesis presents the findings of our research on developing methodologies for aiding verification of a targeted module in large softwares and demostrate our methodologies on a test case. This chapter presents in detail our major contributions. A short summary is as below-

Methodology Our process involves following steps as summarised by the diagram in Figure 3.1.

1. Invariant Mining with beliefs: Mine invariants over the interface of f from simulation traces on S. Invariants capture conditions on input

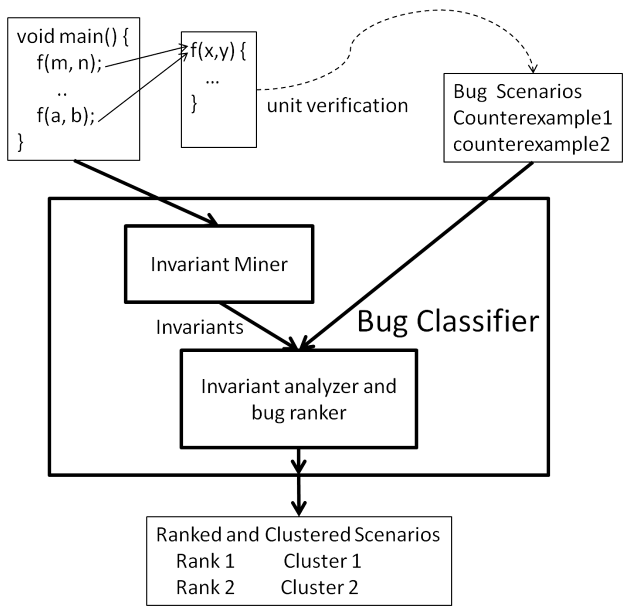


Figure 3.1: *Process Outline*

scenarios fed to f by S, and thus give the details of the environment for f provided by S. Each invariant has an associated confidence of its correctness.

2. Failure scenario Ranking: Failure scenario which contradict invariants of high confidence are given lower ranks.

This work focuses on using dynamic invariants from execution traces for the purpose of ranking. A dynamic detector of program invariants examines variable values captured during execution over execution traces and reports properties and relationships that hold over those values. We use a random input generator or given test suite to generate execution traces on S. These traces are used as input to a dynamic invariant detector Daikon to get invariants on inputs of f. S interacts with f by passing variables as input to f. And so by mining invariants on this interface we get the summary of environment provided by S to f. We propose to use this summary to classify failure scenarios of unit test on f into ones which have a high likelihood of being true and filter out the ones which are likely to be fictitious. Also, we wish to create failure groups that provide a similar environment to f during their course.

3.1 Illustration on the toy example

We consider the simple example in Chapter 1 to illustrate our approach.We want to verify the function int div_by_f(int c, int f). It is expected to compute some non-zero function of c and f and return a value. When return value is 0, it is considered as an erroneous return where the function does not adhere to the requirement. Unit testing isolates each part of the program and shows that the individual parts are correct. If we verify the function div_by_f() in isolation, a unit tester may give a lot of error scenarios. For example, it can give scenarios corresponding to the following evaluation when a call to div_by_f() is made

```
unit-under-test:div_by_f()
1. (c=11,f=2)
2. (c=-189,f=9)
3. (c=10,f=0)
```

As a unit tester has no knowledge of the conditions under which a call to div_by_f() can be made, it might give a lot of scenarios which are not feasible at all. For example scenario 1 is never possible in the current context of the function being called as the environment forces the condition $c\%f == 0$. For non trivial software we cannot find such conditions with definiteness. Our objective is to classify spurious and real error among all the error scenarios presented by a unit tester. We employ dynamic invariants to rule out some of the scenarios. We use the associated confidence of each dynamic invariant to compute the rank of a failure scenario. The two steps summarised earlier are given as applied to the example.

```
//Program S
//Targeted module div_by_f( int c, int f)
  int gcd(int u, int v) {
    int t;
      if(v == 0) v = u;
      while (v) {
       t = u;
       u = v;
       v = t % v;
     }
     u = u < 0 ? -u : u; /* abs(u) */

     return u == 0 ? 1: u;
 }

 int div_by_f( int c, int f){
     int i=0,result;

     if(c >0){
          i = c + 20*f;
          printf(”%d”,i);
      }
      if(i%f != 0)
          return 0;//error
      if(f== 0)
          return 0;//error
      result = i/f;
      if(result < 0)
          return 0;//error

    return result;
 }

 int main(int argc ,char* argv[]){

   int s,t,a,v;

     scanf(”%d %d %d”,&s,&t,&a);

     v = gcd(s,t);

     x = div_by_f(s, v);
    assert(x);

    switch(a)
    {
        case 1:
        x = div_by_f(t, v);
        break;

        case 2:
        x = div_by_f(s, v);
        break;

        case 3:
        x = div_by_f(s+t, v);
        break;

        case 4:
        x = div_by_f(s*t, v*v);
        break;
    }

    assert(x);
  return 0;
}
```

Simple Example

3.1.1 Invariant Mining with beliefs

A program invariant is a property that is true at a particular program point or points. Despite their advantages, invariants are usually missing from programs. An alternative to expecting programmers to fully annotate code with invariants is to automatically infer likely invariants from the program itself. Invariant detection recovers a hidden part of the design space: the invariants that the programmer had in mind. This can be done either statically or dynamically. Static analysis examines the program text and reasons over the possible executions and runtime states. Static approaches are not scalable and hence not used widely in practice. Daikon [7] is an invariant miner. A dynamic invariant detector runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. It does not suffer from the scalability drawbacks faced by static analysis and so complements static analysis. Invariants over the interface of div_by_f() from simulation traces on S are mined using daikon. Invariants capture conditions on input scenarios fed to div_by_f() by S, and thus give the details of the environment for div_by_f() provided by S. Each invariant has an associated belief. Dynamic invariants are mined on the example on inputs c, f of function div_by_f(int c, int f) which are as given in Figure 4.2.

```
=====================================
..div_by_f():::ENTER

f >= 1         confidence = 0.9

c % f == 0     confidence = 0.7
=====================================
```

Figure 3.2: *Invariants on Simple example*

3.1.2 Failure scenario Ranking

Failure scenario which contradict invariants of high confidence are given lower beliefs. These scenarios are then sorted on the basis of given beliefs to give a ranked list of failure scenarios. A scenario having highest belief has the highest rank and should be examined first. We consider each bug scenarios for div_by_f (int c, int f) and consider which invariants it contradicts. We assume the invariants generated by dynamic invariant miner are independent and propose a simple measure to compute the belief. Thus, belief assigned are:

Scenario 1 (c=11,f=2)

When div_by_f() is called it contradicts the invariant $c \% f == 0$ and thus has a low belief. we want to associate a belief to this failure scenario based on what we know of the environment. If this failure has to be true, then the high confidence invariant (reported with confidence 0.7) from Daikon has to be false. Hence, the belief of this scenario to be actually real is calculated to be 0.3 using the following expression-

$$(1 - confidence(c \,\%\, f \;==\; 0))$$

Scenario 2 (c=-189,f=9)

When div_by_f() is called it contradicts none of the invariants and so is likely to be real. We give a belief of 1.0 to scenarios that do not contradict any of the invariants.

Scenario 3 (c=10,f=0)

When div_by_f() is called it contradicts the invariant $c \,\%\, f \;==\; 0$ and $f \;>=\; 1$. This failure can be true only when both the invariants are false. We thus combine the likelihood of both invariants to be false in the expression below and see that this failure has a low belief. In this case the belief evaluates to .03 using the following expression-

$$(1 - confidence(c \,\%\, f \;==\; 0)(1 - confidence(f \;>=\; 1)))$$

Thus, when presenting these scenarios to a developer for debugging, these scenarios will be presented in the sorted order of beliefs.

```
unit-under-test:div_by_f()
Ranked bugs
2. (c=-189,f=9)    belief = 1.0   rank = 3
1. (c=11,f=2)      belief = 0.3   rank = 2
3. (c=10,f=0)      belief = 0.03  rank = 1
```

3.2 Flow of Detailed Methodology

This section gives the detailed methodology of our approach. The problem that we attempt to solve as mentioned earlier is:

Problem We are given $< S, f, TC >$ consisting of:

- S is software in which the targeted module lies.
- f is the targeted module which belongs to the software S.
- TC is the set of bug scenarios on module f. TC contains n number of bug scenarios. Each such scenario gives a faulty execution of f.

Our objective is to:

- Classify and rank the bug scenarios among the test cases to be presented to the developer for correction in decreasing order of rank or belief in the scenario.

We present a ranking and clustering methodology based on dynamic invariants and confidence measures as illustrated in the figure.

The various steps involved, as summarised in Figure 3.3, are-

3.2.1 Invariant Miner

We use Daikon [7] a dynamic invariant detector to mine dynamic invariants. A dynamic invariant miner examines variable values captured during execution over execution traces and reports properties and relationships that hold over those values. In our case we use Daikon to mine invariants on the inputs of the function f to be tested. S interacts with f by passing variables as input to f. And so by mining invariants on this interface we get the summary of environment provided by S to f. We use a random input generator

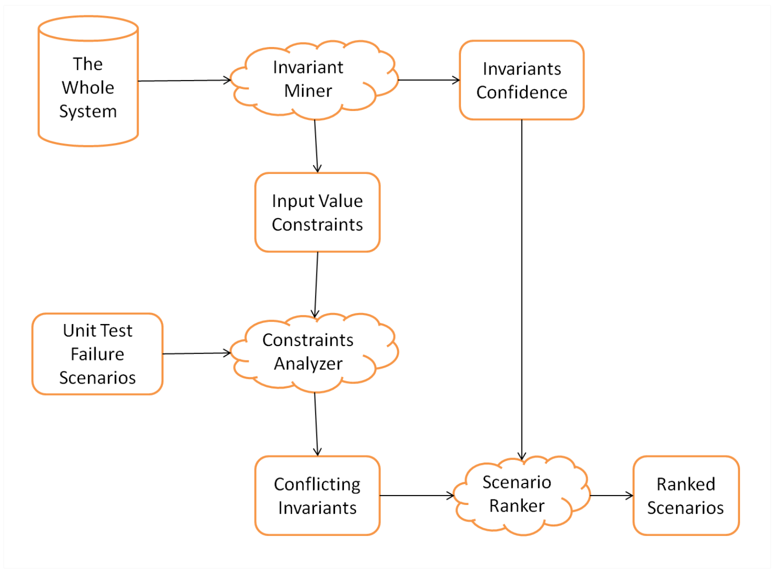


Figure 3.3: *Framework*

to generate execution traces on S. These traces are used as input to the dynamic invariant detector Daikon to get invariants on inputs of f. Let there be m invariants mined.

3.2.2 Confidence

Daikon sets the confidence of each invariant it produces in its output [7]. Each type of invariant has its own rules for determining confidence. For example, consider the invariant $a < b$ whose confidence computation is $1 - 1/2^{numsamples}$, which indicates the likelihood that the observations of a and b did not occur by chance. If there were 3 samples, and $a < b$ on all of them, then the confidence would be 7/8 = .875. If there were 6 samples, and $a < b$ on only 5 on them, the confidence would be 0. If there were 9 samples, and $a < b$ on all of them, then the confidence would be $1 - 1/2^9$= .998.

3.2.3 Constraint analyzer

For each failure scenario we need to examine m invariants to rank it. We identify all the invariants a failure scenario dissatisfies among these m invariants. A failure scenario dissatisfying a huge set of invariants should be ranked lower as opposed to a failure scenario which is in agreement with all the invariants. The idea behind this is that invariants serve as our eyesight to the environment provided by the software. If a failure scenario is not in agreement with this environment then it is possibly a false failure scenario which is not feasible. The additional issue to be addressed when using dynamic invariants instead of static is, although dynamic invariants hold across all the traces from which they are mined, a dynamic invariant may not hold across an execution which is not yet seen. To address this issue, we use the confidence measure Daikon associates with each dynamic invariant, to compute the rank of a failure scenario.

3.2.4 Scenario Ranker

For each scenario there is a set of invariants contradicted by it. We initialize the belief of the scenario by 1. For each invariant in the set of invariants contradicted we multiply the belief by one minus the confidence of that invariant. We adopt such a method under the following reasoning. A failure scenario on f contradicting a set of invariants is a true failure iff there exists a scenario in S which witnesses the failure scenario on f. Such a scenario in S allows the values taken by the scenario on f. As the failure scenario on f contradicts the given set of invariants, so does the scenario. Thus, when the function f is called (or where the invariants are mined) all the invariants the failure scenario contradicts are false simultaneously. As the invariants do not hold at this point in the scenario, these all should be false invariants.

If an invariant i is true with confidence c, we say that, it is false, or its negation might hold at some point, with confidence $1 - c$. That is $confidence(i\, not\, true) = 1 - confidence(i)$ Next we want a confidence on $i_1\&\&i_2$ given confidence on i_1 and confidence on i_2. If these invariants are independent we can easily multiply their beliefs.

```
Data: UI = unsatisfied invariants set for a failure scenario
Result: Belief of the failure scenario
belief = 1;
for each i in UI do
  | belief = belief*(1 - confidence of invariant i);
end
```

Algorithm 1: Compute belief of the failure scenario

This way, beliefs are computed for each scenario. The scenarios are then sorted in decreasing order of beliefs.

Chapter 4

Applications

In this chapter we present the various contexts where this thesis might prove helpful in making some improvements. Before that we develop on the methodology itself in the earlier chapters by adding a way to group failure scenarios before ranking them. Thus, we present two applications of the idea in bug prioritization and localization.

4.1 Clustering and prioritization

We develop on the methodology presented in Chapter 3 by adding a way to group failure scenarios before ranking them. Given a set of failure scenarios on a function f, when ranking them on their occurrence in reality, we would also like to minimize the scenarios to be actually examined by the developer manually. We would not like a developer to examine the manifestation of a single error more than once. We would like to present all such scenarios in a group to be examined at once as these may be related. We would like to save manual effort and try to give error scenarios which point out more bugs in less number of examinations done by a developer.

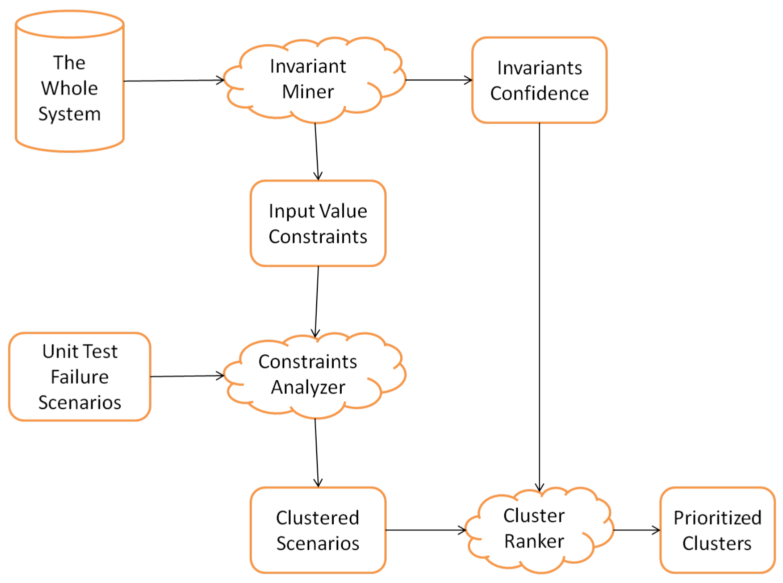


Figure 4.1: *Framework Application*

Clustering For each failure scenario we need to examine m invariants to rank it. We could also use the same idea to group or classify the failure scenarios into groups. A modified framework is presented in Figure 4.1. All failure scenarios dissatisfying the same set of invariants are put in the same group, thus forming clusters of the failures. The idea behind this clustering is that if we take care of one failure scenario from this group, the other failure scenarios might be taken care of in the process. That is, they might be manifestations of the same error or may be related. These may have arisen due to a certain environment, in which our unit-under-test was incapable of functioning correctly. Once these conditions or environment is taken care of, the whole cluster of errors will vanish.

Prioritizing For each cluster there is a set of invariants contradicted by the failure secnarios in it. We initialize the belief of the cluster by 1. For each invariant in the set of invariants contradicted we multiply the belief by one minus the confidence of that invariant as done by Scenario Ranker. We adopt the same method under the same reasoning. We just can do the calculations once for finding the belief for the cluster instead of for every failure scenario.

```
Data: UI = unsatisfied invariants set for a cluster
Result: Belief of the cluster
belief = 1;
for each i in UI do
    belief = belief*(1 - confidence of invariant i);
end
```

Algorithm 2: Compute belief of the cluster

Also something to notice is, a failure scenario cluster $c1$ in disagreement with $I1$, a superset of invariants with which $c2$ conflicts say $I2$, will have a lower confidence. This makes sense as it may be possible that the invariants in $I1 - I2$ have no significance in the error in $c1$, and that failure scenarios in $c1$ are a manifestation of the same error as $c2$.

Clustering and prioritization There are 2^m possible clusters on the basis of whether a failure scenario satisfies or dissatisfies a given invariant. For each failure scenario we need to examine m invariants to classify it into such a group, thus requiring $O(m)$ time for each failure scenario. For n failure scenarios we need $O(mn)$ time. We examine the invariants with higher confidence first. As the invariants are examined one by one for a given failure scenario its belief decreases. Therefore, if we can use a cut off belief, then we need not examine the other invariants for failure scenario having a belief already below the cut off, we can simply drop the failure scenario if its belief falls below cutoff at any time.

4.2 Localization

Software is far from being bug-free. Manual debugging is laborious and expensive specially in large software. Bug localization is to find a set of source code locations that are likely buggy through automatic analysis. In general we are given a set of failing executions, and a set of passing executions. For the failing executions we would like to know the location of bug that caused the failure. This task is known as bug localization [16], which is defined as a classification problem: given n source code entities and a bug report, classify the bug report as belonging to one of the n entities. The classifier returns a ranked list of possibly-relevant entities, along with a relevancy score for each entity in the list. An entity is considered relevant if it indeed needs to be modified to resolve the bug report, and irrelevant otherwise.

We create a very simple example to illustrate how our approach can be used for bug localization as in the code listed below. Consider we want to verify the function greater().

Unit testing isolates each part of the program and shows that the individual parts are correct. We could use a unit tester such as CUTE or KLEE. It might give a lot of scenarios where error label is reachable. We have already suggested a way to classify spurious and real error among all the error scenarios presented by a unit tester.

```
Program S
Targeted module greater(s,t)

  int local_1(int s,int t){
     return 0;}
  int local_2(int s,int t){
      return 0;}

  //Expected to evaluate and return s>t
  int greater(int s,int t){
      //point r:implementation error
      return s>=t;
      //logical error allows equal values
```

```c
}

int main(int argc ,char* argv[]){

    int s,t,seed,range;

    range = 100;
    seed = atoi(argv[1]);

    srand(seed);

    s = rand()%range;
    t = rand()%range;

//location1:
    if(s==t){
        printf(”s=t”);
    }
    else
    {   //point p:no error
        local_1(s,t);
        if(greater(s,t))
            printf(”s>t”);
         else
         {    //point q:no error
              local_1(s,t);
              if(!greater(s,t))
                 printf(”s<t”);
         }
    }

//location2:
    //point e:error
    local_2(s,t);
    if(greater(s,t)) //erroneous for s==t
        printf(”s>t”);
    return 0;
}
```

Simple Example 2

4.2.1 Our Method

Dynamic invariants mined on the example on relevant function inputs are as given in Figure 4.2. We consider two alternative locations where a function call to greater() needs to be examined in different contexts. It is expected to perform a check if $s > t$ and return the boolean evaluation. Given two locations in the software S and a bug report in the form of a set of failing executions, we want to classify these failing executions as belonging to one of the two locations.

```
locations from where greater() is called:

1.      location1:
2.      location2:
```

We consider a failure scenario evaluation $s = 4, t = 4$ and examine from which location a failure of this kind is possible.

```
===============================================================
..local_1():::ENTER <---constraints for case 1

  s != t         confidence = 0.9  <---invariant contradicted
  s >= 0         confidence = 0.9
  t >= 0         confidence = 0.9

..local_2():::ENTER <---constraints for case 2

  s >= 0        confidence = 0.7
  t >= 0        confidence = 0.9
===============================================================
```

Figure 4.2: *Invariants on Application example*

location 1 greater() is called with inputs constraints specified under side heading $..local\_1() ::: ENTER$. This failure scenario contradicts the invariant $s! = t$ and thus has a low belief. In this case $(1 - confidence(s! = t))$, which is 0.1

location 2 greater() is called with inputs constraints specified under side heading $..local\_2()\ :::\ ENTER$. In this case the failure scenario does not contradict any of the invariants listed and has a belief of 1.0

```
Order in which locations will be examined
for cause of failure:

      location          local belief of failure

1.    location2:           1.0
2.    location1:           0.1
```

As the scenario has a higher belief in location 2, the scenario has possibly generated from this location. Thus, when listing the possible locations for the origin of this error, location 2 is listed above location 1.

Chapter 5

Implementation

5.1 System Architecture

The architecture of our approach mainly includes the following components: invariant mining, confidence measure, clustering and ranking. Our implementation takes a software S, a targeted module f, which lies in the software and a set of available test cases for the software. It outputs a report which contains multiple clusters of failure scenarios or bugs in the targeted module f, ranked in decreasing order based on their likeliness of occurrence in the software life cycle.

5.2 Failure Extraction

To produce the likely bug scenarios of the module f, we use CREST, a concolic unit testing engine in C. Crest [1] is an open source Concolic Testing Engine for C, a reimplementation of CUTE (Concolic Unit Testing Engine) [15]. It uses CIL (C Intermediate Language written in Ocaml) to insert instrumentation code into the given program and perform symbolic execution in parallel with concrete execution to explore all feasible program paths. It uses Yices (an SMT solver) to solve symbolic constraints and generate inputs which enable CREST to infiltrate into new paths. At present it supports only linear and integer arithmetic and has no support for pointers/dereferences and bitwise operators.

For testing with CREST we need to include crest.h in the target program and use CREST_type(x) to mark symbolic variables, where type can be int, short, char, unsigned_int, unsigned_char or unsigned_short. Then crestc is to be run with the target source code to enable CREST to perform instrumentation. After this run_crest will run the program executable with a search strategy. CREST provides 5 search strategies: dfs depth first search, cfg nearest uncovered branch first, random-negated branch randomly selected, uniform_random and random_input. By default CREST produced one input file and one coverage file which contain the last input and branch coverage information. We tweaked the source code of CREST in our work to make CREST output all input combinations and coverage information for each run in a separate file.

5.3 Invariant Mining

Now we need to estimate the environment provided by the software S to the specified module f. In other words, in this step we figure out the boundary value conditions of the arguments passed to f by the software while running over the available test cases. We implement this step using the kvasir front-end of Daikon. Daikon [7] is a dynamic invariant detector which reports likely program invariants in C, C + +, Java, and Perl programs, and in record-structured data sources. It is easy to extend Daikon to other applications. Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data. It is to be noted that the invariants proposed by Daikon are justified by execution trace of the program. It generates only those invariants whose confidence is above the threshold value set and outputs the confidence associated with each invariant so produced. We use the program invariants produced by Daikon to cluster the test cases first based on which invariant conditions conflict a particular failure scenario. Examples of invariants include being constant $(x = a)$, non-zero $(x \neq 0)$, being in a range $(a \leq x \leq b)$, linear relationships $(y = ax+b)$, ordering $(x \leq y)$, functions from a library $(x = fn(y))$, containment $(x \epsilon y)$, sortedness $(x \; is \; sorted)$, and many more. Several considerations determine the set of variables over which the core Daikon inference engine operates when it infers the invariants for a given execution trace of a program. First, the set of variables must be large enough to enable the inference of meaningful invariants whose enforcement can meet our objective. Second, the set must be small enough to make the inference task computationally tractable. Finally, the values in the variables must be defined in all possible executions (and not just the observed executions).

For each test case available we first run the kvasir front-end tool (also known as instrumenter or tracer) on the software S which results in production of separate $.dtrace$ and $.decls$ files for each test case. Before using kvasir, the software has to be compiled with the $gdwarf-2$ flag enabled to produce $DWARF-2$ format debugging information along with the program. A $.dtrace$ file contains information about a particular execution of the program, the values of the program variables at each program point. A $.decls$ file consists of the information about what variables and functions exist in a program, along with information grouping the variables into abstract types.

Next we use java $daikon.Daikon$ to produce a single $.inv$ file which contains all invariants found over all the execution traces in binary format. There are numerous control, optimization and debugging options available with Daikon that have been used to produce suitable invariants for our purpose. The confidence threshold was set to 0 so that we can work with all kinds of invariants. For our program, we focused on only the targeted unit module and mined invariants over the program points corresponding to the same.

In this step we used a simple shell script to run kvasir over the available test cases of software S and run $daikon.Daikon$ over all the traces in a single step along with numerous control and optimization options. We use the $.inv$ file thus produced in our next steps.

5.4 Confidence Measure

The confidences reported by Daikon depend on the kind of invariant involved. Each type of invariant has its own rules for determining confidence. The details of each invariant along with its confidence is stored in $.inv$ file produced by Daikon.

5.5 Clustering of Failure scenarios

Now we have n failure scenarios which embody a set of possible bug scenarios for the target module f, where f does not follow some given specifications or violates some runtime conformations (for example, Segmentation fault, buffer overflow error, stack smashing error, divide by zero error, etc.) and m invariants, each assigned a confidence as per the measure discussed previously. For each failure scenario we prepare the set of invariants which it conflicts with. Now we group all such failure scenarios which conflict with the same set of invariants into a cluster. The underlying theory of the approach is that the failure scenarios, which conflict with the same set of invariants, simulate the same kind of environment which is more likely to produce a program failure. Hence grouping them into the same family, we might be able to infuse more diversity into the failure scenarios, also leading to faster and more efficient bug detection for the programmer.

To implement this step we have made the use of Daikon's library through a code snippet written in Java. In this code, we used Daikon as an API for our purpose. We parsed the $.inv$ file, which contains a $PptMap$ object in serialized form, produced by the previous step to extract all invariants and all information about them for evaluation of confidence.

5.6 Ranking

At this step we finally have a set of families of failures, each having a single or multiple failure scenarios. Now the developer will want to know whether all such families are equally likely to occur in the software life cycle. To address this issue we aim to rank or prioritize these families and produce a more organized summary to the developer. As all failure scenarios of a family contradict the same set of invariants, we can choose a single member of each family for our evaluation. To cater to our needs we initially assign a rank 1 to each cluster. Then for each invariant, in the set of invariants contradicted, we multiply the rank by one minus the confidence of that invariant. The effect of this being that if a class of failures conflicts with a low confidence invariant then its rank is increased as its likeliness of occurrence is more. Again if it conflicts with a high confidence invariant then it is less likely to pose a problem in the software life cycle, hence its rank is reduced. Finally after this process we have a set of bug families of the target module f, prioritized on the basis of their chances of interfering with normal execution of the software S.

For the purpose of clustering and ranking we designed a simple Java program which takes the failure scenarios and invariants as inputs along with their confidences and produces the set of clusters arranged in decreasing order of relevance.

Chapter 6

Evaluation

We now report our experience in using our method for clustering and ranking failure scenarios on the replace program written in C from the Siemens Benchmark Suite.

6.1 Experience with replace

We describe our experience with the replace program, the largest program of the Siemens Benchmark Suite. The Siemens Researchers [10] produced a test pool which ensures that each exercisable coverage unit was covered by at least 30 different test cases where two test cases are considered different if the simple control paths that they exercise differ. They provide 5542 test cases for the replace program, 1 base version of the program and 31 faulty versions, each differing from the base version at some points. For our purpose we used a single version of the program and checked for runtime violations of all the functions available in the program. Henceforth we report the application of our approach to this benchmark.

6.1.1 Testing with CREST

Firstly we need to generate the failure scenarios over a unit level function in the program. The replace program searches for the occurrence of a given string in a file and replaces it by another given string and displays the result to standard output. We tested the function $dodash()$ in the replace program and found that it contains some stack smashing errors for particular combination of inputs. The function $dodash()$ has 5 parameters: two integer pointers, two strings and one character. For testing with CREST we declare all these variables as symbolic using $CREST\_type(x)$ where type is the variable data type. We fix the length of two strings at 5 for this setup as CREST cannot symbolically create strings of arbitrary lengths. We also need to include "$crest.h$" to enable crests features.

```
main()
{
    char delim;
    char src[5];
    int i;
    char dest[5];
    int j;

    CREST_char(delim);
    CREST_char(src[0]);
    CREST_char(src[1]);
    CREST_char(src[2]);
    CREST_char(src[3]);
    CREST_char(src[4]);
    CREST_int(i);
    CREST_char(dest[0]);
    CREST_char(dest[1]);
    CREST_char(dest[2]);
    CREST_char(dest[3]);
    CREST_char(dest[4]);
    CREST_int(j);
    CREST_int(maxset);
      dodash(delim, src, &i, dest, &j, maxset);
}
```

We compile the code using CREST using the $crestc$ (crests compiler).
$crestc\ replace.c\ o\ replace$
Then we run run_crest over the executable file thus produced.
$run\_crest\ ./replace\ 1000\ dfs$
Here 1000 refers to the maximum number of iterations that CREST must use during path exploration using the depth first search strategy (specified here by dfs flag). CREST by default produces a single input and coverage file with the information about the last iteration only. We edited the source code of CREST to make it produce an "$inputs$" and "$coverages$" file which contain all inputs and branch coverage information produced during each iteration of CREST. In 34 of these iterations the program produced stack smashing errors and the program was terminated. We extracted these 34 set of inputs from the inputs file and use them as failure scenarios for our work. We use these inputs for our methodology as failure scenarios with n=34.

6.1.2 Invariant Mining

Now after we have the failure scenarios, we need to mine invariants over the function $dodash()$ with the help of Daikon. We use the test cases provided by the Siemens benchmark suite to train Daikon. Out of the 5542 test cases for the replace program we use the first 94 test cases.

Kvasir We used the kvasir front end tool to produce $.decls$ and $.dtrace$ files for each run of the program. we had to make use of pointer type disambiguation to enable Daikon to produce proper invariants. For invariant mining in Daikon and Kvasir, it is necessary to specify whether pointers refer to arrays or to single values and optionally also specify the type of pointer. For example, in the following sample code snippet,

```
 void sort(int *arr,int *x,char *m,char *str){  }
 //definition of sort
...
int my_arr[10];
int p=5;
char m=a;
char s[]=string

sort(my_arr,&p,&m,s);      //use of sort
```

Here arr is an array of integers but x is a pointer to an integer, m is a pointer to a character but str is a string. So we must explicitly specify during invariant detection what each such pointer corresponds to. We must specify them in a separate file before using Kvasir. Kvasir has a $--disambig$ option which allows it to read such a file and use it while evaluating and producing meaningful invariants.

Daikon Next we used Daikon to mine invariants on all these trace and declaration files combined. We used the following options along with Daikon to enhance the quality of invariants:
$conf\_limit\ 0$ this sets the confidence threshold to 0. So basically Daikon reports back all kinds of invariants found.
$ppt-select-pattern="dodash*"$ this makes Daikon report only the invariants found over the program points corresponding to our target function $dodash()$
$config\_option\ daikon.Daikon.print\_sample\_totals = true$ this option makes Daikon print the total number of all kinds of samples(variable values) found during its operation.

$java\ daikon.Daikon\ --conf\_limit\ 0\ --ppt-select-pattern = "dodash*"\ --config\_option$
$daikon.Daikon.print\_sample\_totals = true-oinv\_files/replace.inv.gz$
$kvasir\_output/replace\_*.dtrace$
The result of this is a $replace.inv.gz$ file which contains all invariants found at the entry and exit points of the function $dodash()$ over all executions in serialized format. Hence the process of invariant mining is completed. While assigning confidences to the invariants and filtering, it is to be noted that not all invariants found are useful for relative ranking and clustering. The invariants which do not conflict any failure scenarios are assigned a confidence of 1. These invariants are not useful for the stage and hence are not used. The invariants which conflict all failure scenarios are also not relevant to our work and are not used. This helps improve the performance of our ranking methodology. After the filtering process, we found 4 invariants to be useful out of the 15 reported by Daikon. We use this set of invariants for our methodology with m=4.

6.1.3 Confidence Measure and filtering

Number of calls made to the function $dodash()$ are 152 in 94 test cases. The confidences found for each invariant is shown in the table below.

inv no.	invariant c	unique values on invariant $un(inv\ c)$	confidence
2	delim == 93	1	1.0
7	j[0] != 0	42	0.0
8	maxset == 100	1	1.0
14	i[0] != j[0]	152	1.0

6.1.4 Failure scenario Clustering and Ranking

There were 5 different clusters formed as a result of our clustering process. The order in which these would be presented to a developer is as shown in the table below.

cluster no./ order	invariant contradicted by the cluster/(inv no.)	size of cluster/(no of failure scenarios)	$belief$	rank
1.	2, 7, 14	4	0.0	1
2.	2	3	0.0	1
3.	8	11	0.0	1
4.	2, 8	5	0.0	1
5.	2, 7, 8, 14	11	0.0	1

We see that the beliefs of all the clusters thus formed are 0. Therefore, it is very likely that the failure scenarios in this case are all spurious. Seemingly the confidences assigned by Daikon are not useful in this case to rank the clusters. The only advantage we get here is we present clustered failure scenarios, and developer can examine failure scenarios within a cluster together as they may be related to each other, and might help reduce debugging effort.

Chapter 7

Conclusion and Future Work

In this work, we have presented a methodology to rank and prioritize bug scenarios on a unit module in a software. Our framework, given a set of failure scenarios, uses invariants to classify these bugs into bug families. Our technique combines unit testing or model checking of a functional unit with use of dynamic invariants. Our experience with real-life case study (replace program, the largest program of the Siemens Benchmark Suite) demonstrates the utility of our method.

For our purpose we implemented our own confidence measure over the available invariants based on the number of failure scenarios with which they conflict. We have assumed the invariants to be independent throughout. For future work, we may try to evaluate confidences of combined invariants without assuming independence to get better results.

Bibliography

[1] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169. Springer, 2000.

[3] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking. In *NATO ASI DPD*, pages 305–349, 1996.

[4] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative data-flow analysis, revisited. Technical report, 2004.

[5] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[6] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Software Eng.*, 28(2):159–182, 2002.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[8] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, pages 634–640, 2009.

[9] C. A. R. Hoare. Assertions. In *IFM*, pages 1–2, 2000.

[10] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[11] C. B. Jones, P. W. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[12] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[13] S. Mitra, A. Banerjee, and P. Dasgupta. Formal methods for ranking counterexamples through assumption mining. In *DATE*, pages 911–916, 2012.

[14] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–188, 1999.

[15] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[16] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2013.