Interactive Specification Mining for Debugging Embedded Software Systems

by

Taha R. Siddiqui

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
July 2017

Thesis advisor

Dr. Hadi Hemmati

Author

Taha R. Siddiqui

Interactive Specification Mining for Debugging Embedded Software Systems

Abstract

Specification mining techniques are typically used to extract the specification of a software in the absence of (up-to-date) specification documents. Several research projects have targeted the problem in the past. However, very limited application of such techniques is observed in industry, due to challenges related to accuracy and practicality of those techniques. Such specifications are useful for program comprehension, testing, and anomaly detection. However, specification mining can also be used for debugging, where a faulty behavior is abstracted to give developers a context about the bug and help them locating it. In this thesis, I proposed, developed, and evaluated an interactive semi-automated specification mining approach that not only helps generating targeted and correct specifications of a system but can also be used specifically for debugging. The tool users can select relevant state fields and functions, per issue, interactively, and run the tool on a reported faulty scenario. The tool generates a state machine that represents the faulty behavior, abstracted with respect to the users' inputs. These models are then used to locate the fault's root cause for debugging. I have applied the technique and tool on an AutoPilot software system for UAVs, from Micropilot Inc. I evaluated the approach and tool in a set of

experiments, based on Micropilot's actual reported issues. I interviewed Micropilot developers after they used the tool in an experimental setup for debugging different real issues and collected their feedback. The results have shown that the approach is feasible, and brings advantages over only using code-level debugging tools.

Acknowledgments

Firstly, I want to take this opportunity to thank my supervisor, Dr. Hadi Hemmati, for his continuous support and guidance for the entire duration of my Master's program at the University of Manitoba. I am grateful to the Department of Computer Science, University of Manitoba for providing me the financial support through Guaranteed Funding Package (GFP) so that I could keep my focus on my thesis project and consider it as a full time job.

Secondly, I would like to thank Mr. Asa Indrabudi from Micropilot Incorporation for assisting me in understanding the projects and processes at Micropilot Inc.

I would like to thank my co-advisor Dr. Rasit Eskicioglu, and the committee members, Dr. James Young and Dr. Bob Mcleod for their careful review and valuable feedback.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the increasing applications of Unmanned Air Vehicles (UAVs), e.g., in military, cargo, agriculture, and surveillance domains, safety issues related to the operation of UAVs have become more important than ever. One of the safety considerations of such systems is their software safety. Aviation software industry observes one of the highest standards of safety control, where their embedded software systems go through a set of rigorous standard checks, before entering the commercial market. It is so because even a short-time failure in an air vehicles controller software could be life threatening or cause extensive damages. Therefore, it is crucial to verify the software systems'correctness, robustness and reliability. Federal air certification authorities, such as Transport Canada in Canada and Federal Aviation Authority(FAA) in the United States, use a standard document, Software Considerations in Airborne Systems and Equipment Certification (DO-178C), to certify the commercial use of an airborne software system. One of the very critical features of this document is making sure that the software has an explicit and a complete set of system-level re-

quirements that are verified by a set of test cases. However, with the advent of fast paced programming (such as Agile methods), there is lesser time for the software development teams to maintain an up-to-date set of requirement documents. Moreover, for the companies which are in this business for a long time when the standards were not strictly defined, attaining this certification is even more costly since they have to create the requirement for the software that is working fine for decades.

In general, software specifications and requirements are considered very important during several stages of the software development life cycle. First, they are helpful to developers for developing the system, for testing it effectively and releasing a foolproof system. They are also helpful for users(clients) to understand the software.

Although the specifications differ for different kind of users, they are depictions of the very same system in different level of details. They can even be presented in different forms; text, figures, formulae, or any other form that can show program structure, flow, or behaviour easily. However, building specifications (models or documents) manually and keeping them up-to-date are very costly. Hence, there is a need for automated techniques for software specification generation.

The area of specification mining is not new and hence there are many studies that have focused on the problem of automated extraction of software specification as in [25]. The techniques, in general, apply reverse engineering to abstract specification of the system from the source code, execution traces, logs files, etc. Broadly speaking, the techniques can be categorized in either static or dynamic analysis. In static analysis program artifacts such as source code are analyzed without actual execution, but real executions are needed to infer the behavior of software in dynamic analysis.

While the main application of generating specifications of the system is program comprehension, they can also be used for other purposes as well, including requirement document generation, program monitoring, debugging, and automated test generation. In my research, I have targeted the domain of debugging application with the help of specification mining, where I exploit the capabilities of execution traces to collect as much information during an execution, and present it in an abstracted and organized way for developers to have a consolidated but detailed view of the program.

The application domain of the study is safety critical systems, however, it can be applied to any real world $C/C++$ software repository. The research was in collaboration with the industry partner of Software Engineering and Analytics Lab (SEALab); MicroPilot, Inc., a leading manufacturer of autopilots for Unmanned Aerial Vehicles (UAV) and Manned Aerial Vehicles (MAV), with a clientele of over 850 in 70 countries in the area of academic, military and private research. The code base used in this project UAV autopilot software, working on different kinds of vehicles, including fixed wing, multi-rotor, blimp and boat, is developed by MicroPilot, Inc. The repositories are maintained as several Visual Studio solutions, with several modules running as Visual Studio .NET 2008 and others as Visual Studio .NET 2015.

This study is part of a bigger project that deals with certification of safety critical systems. The specification mining solution presented in this thesis will help the company acquiring the certification from different aspects, e.g., requirement generation, debugging, and model-based testing. However, this thesis focuses only on the debugging and program comprehension part of the project.

Many studies in the past have proposed automated techniques for specification

mining and debugging. In real-world, however, finding the best level of abstraction and the best perspective for abstraction, automatically, is very hard, if not impossible, due to the required domain knowledge that is very hard to be learned by the algorithms. Hence, I propose the idea of a semi-automated approach for specification mining, where the developers specify the perspective (relevant system variables and methods) and level of abstraction (types of constraints they wish to monitor) for the abstraction of system behavior from real execution traces. The approach includes three main steps:

1. Developer selects a list of important fields and functions from an extracted list, and define the constraints to monitor and generate the states in the state machine.

2. Selected fields and functions are traced from the real execution of the program.

3. Execution traces are abstracted in the form of state machines using the defined constraints.

The outcome of this project is a tool developed as a web application. For evaluation purposes, I incorporated the tool with the environment of my subject code-base. It can also be linked and used with other similar embedded systems with automated scripts and minimal configuration changes.

Unlike previous techniques, the interactive nature of the approach lets the user generate state machines of different levels and perspectives, in run-time, using the same execution traces. For instance, making a high-level state machine for program comprehension or zooming into a single high-level state and create a detailed

state-machine for debugging purposes, such that just by changing the definition of constraints, we can obtain different levels of state machines, for different types of users, from the same execution traces.

To evaluate my technique, I have applied the approach to a case study with over 1.3 MLOC in a real safety critical industry setting. I also conducted a set of experiments with eight beginner to expert level developers at the company who were familiar with the code-base to some extent. Some of them were provided with the state machines for debugging a set of real bugs, while the others were asked to use the tool for generating the state machines by themselves and inspect them. This is explained in more detail in the Empirical Study. During the experiment the subjects were also observed to analyze the cost/benefit analysis of the approach and the tool, while after the experiment, they were interviewed to get their perspectives on the usefulness of the tool in the context of software debugging. The results showed that:

- Abstraction of execution traces generally makes it easier for developers to understand the problem and to explain it to their peers.
- State machines are helpful in getting the big picture for an execution, which is useful for the developers to dig in and find the root-cause of an issue.

It should be noted here that the framework is proposed to be used as an additional tool to the main debugging tool rather than a replacement. The developers stated that it would be interesting to have another perspective of the execution and if provided the option they would use it.

The rest of the thesis is divided into the following Chapters.

Chapter 2 describes the motivation of this research and also outlines the contribution made in the area of specification mining, which also maps to the limitations discussed in the same section.

Chapter 3 explains the background of the area of research. It also explains different kind of inference techniques and reviews the related work in the area of specification mining that is close to the approach used in this research. It mainly focuses on the technique that use both functions and data to model a software system. The chapter also discusses the limitation of each of the related work.

Chapter 4 describes and explains my approach in detail. In this chapter I explain 1) the instrumentation method used, 2) a new novel method of state abstraction and state merging, and 3) the scale-able tool that bundles everything in this research in one web application.

Chapter 5 discusses and reports the result of the evaluation of the approach and the tool that has been performed through a round of experiments and interviews from professional developers.

Chapter 6 finally provides conclusion, on the basis of evaluation, as an outcome of this research. This chapter also discusses the limitation of the approach and also present potential improvements that can be applied to the approach in the future, and other possible directions for the project.

Chapter 2

Motivation and Problem Description

In this section, I will explain three main limitations of most existing approaches for specification mining, which serve as motivation for this project.

1. In most previous approaches, a "state" is either defined as an intermediate state of the system between two function calls, or by invariants which are automatically generated and are not precise.

2. Most previous approaches, especially those that extract invariants automatically, require a large amount of learning data, e.g., test executions, and hence are costly in real-world applications.

3. In most previous approaches, it is not possible to extract a partial specification with a focus on specific aspects of the system, for example the context of a bug.

To explain these limitations, let's take an example of a generic autopilot software

and demonstrate the system behavior as depicted by a typical automated specification mining approach and compare it with my proposed approach.

As shown in the code snippet below, assume that a sequence of functions {`accelerate,takeoff`} is called in a loop between the states of `Onground` and `Takenoff`. The `Onground` state is defined as `altitude` <0 and `Takenoff` as `altitude`≥ 0. There might be many function calls that start from `Onground` but change it to `Takenoff`. In the above example, the `takeoff` function, for instance, is called during the `Onground` state without any effect to the state, until the condition `speed`$\geq$ `takeOffSpeed` is true.

```
1  void main(){
2      while(altitude<desiredAltitude){
3          accelerate();
4          takeOff();
5      }
6      while(!flight_success)
7          fly();
8      while(altitude>0){
9          decelerate();
10         land();
11     }
12     park();
13 }
14 void accelerate(){
15     speed+=SPD_INCREMENT_CONST;
16 }
17 void takeOff(){
18     if(speed>takeoffSpeed)
19         while(altitude<desiredAltitude){
20             altitude+=ALT_INCREMENT_CONST;
21             if(altitude>safeAltForGearRetract)
22                 retractLandingGear();
23         }
24 }
25 void decelerate(){
26     speed-=SPD_INCREMENT_CONST;
27 }
28 void land(){
29     if(speed <= landingSpeed)
30         while(altitude>0){
31             altitude-=ALT_INCREMENT_CONST;
32             if(altitude<=safeAltForGearRetract)
33                 deployLandingGear();
34         }
35 }
```

Listing 2.1: Example Code Snippet

Following a traditional state merging strategy, the same function calls will be merged correctly into one transition but there will be two separate states for the `Onground` state, one after each call of `Takenoff` and one after each call of `accelerate`. The partial state machine in Fig 2.1 (the one above) shows this merging scenario using traditional specification mining (where new function calls will result in new states).

To avoid the limitation mentioned above, in this thesis, I have defined an approach in which a state is defined by a set of user-defined `constraints`, over system variables (state fields). Therefore, a new function call will not necessarily affect the state. Fig 2.1 shows the results of my interactive merging strategy, which is explained in detail in the methodology section (Section 4). The main idea is to define the states as above, which reduces the total number of states in the system, and merge the states based on user-defined constraints. Note that this approach can correctly and accurately abstract models even from single execution scenario (one focused aspect of the system), but the invariant-based approaches require several execution results to be able to infer a pattern (motivation 2).

Finally, with my interactive approach the user can generate focused specifications (motivation 3). The example state machine shown in Fig 2.2 is generated using a set of four constraints and the generated state machine focuses only on the "Take Off" aspect of the flight. However, Fig 2.3 abstracts the same execution trace as a high-level flow of the execution.

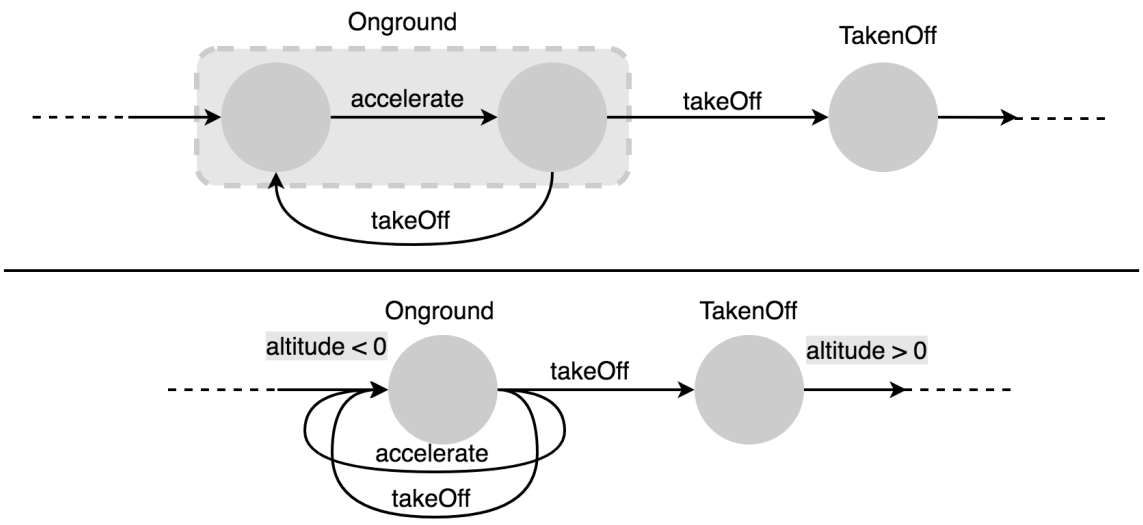


Figure 2.1: Sample extracted models using existing automated approaches (above) vs. my interactive approach (below).

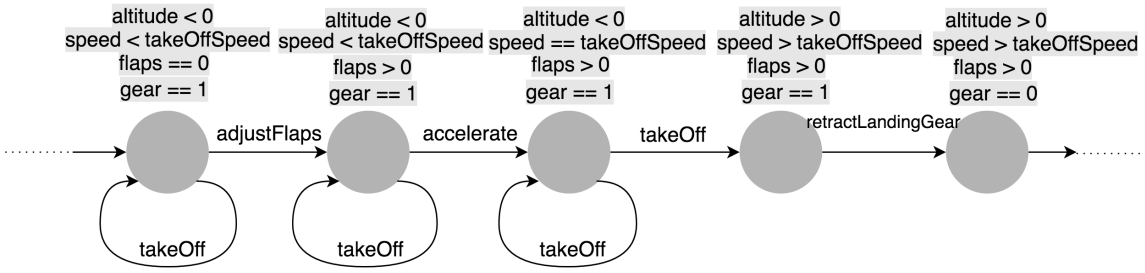


Figure 2.2: Focused state machine on Takeoff

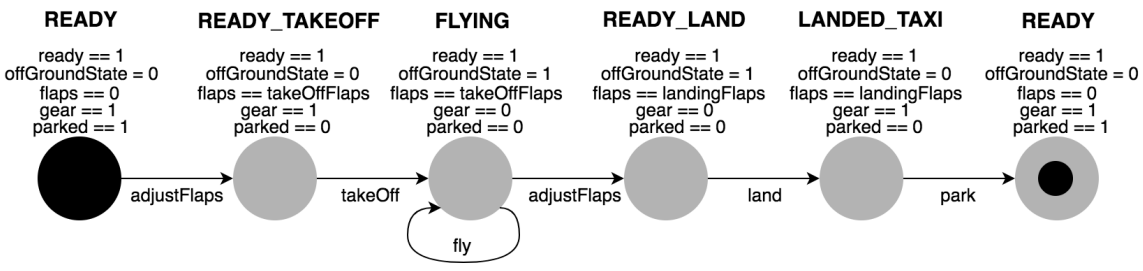


Figure 2.3: High-level state machine of a flight

Chapter 3

Literature Review

There are two methods for program inference to extract program behavior. Active Inference and Passive Inference. In active inference, some form of validation is required actively; such as human or test oracles, to execute the tests on the system and to validate and correct the abstracted models. However, in passive inference, source code is used to either be analyzed or executed for getting execution logs, which are used with state merging techniques to generate abstracted program behavior. This research lies in the area of passive inference, the core of which is state merging algorithm which helps in reducing the number of states in a state machine, hence making it readable. Hence in this method, I reverse engineer the code base by executing the code and abstracting execution traces to generate specifications.

In the past, several research projects have worked on the problem of specification mining through reverse engineering [41; 8; 18]. The two major approaches for this are static and dynamic analysis. Briefly described, static analysis approaches use the source code or other artifacts as is, without execution. On the other hand, dynamic

analysis approaches work by executing the code and mine the specifications, for instance from execution traces. Mining specification from execution traces typically requires instrumentation of the source code to get logs from real execution. Several techniques from both of the approaches are discussed in this section below.

3.1 Specification Mining Techniques

This subsection briefly summarizes the background and related work around dynamic specification mining, which is the context of this study.

3.1.1 Static Analysis

In [17], Gruska et al. introduced an approach to mine temporal rules in the form of Computational Tree Logics (CTL) from program code by static analysis. They built a prototype named Tikanga which takes program source code and a set of CTL templates as input for static analysis.

Using intra-procedural static analysis, they first reveal an object usage model that represents how an object gets used across the whole program. Each of those extracted models is then represented by a Kripke structure. A model checking procedure checks the validity of generated CTL formulas from Kripke template. Using concept analysis technique, they mined each formal argument of a function and associated CTL formulas that satisfy for object passed as the actual parameter. The final output of the procedure, CTL formulas can be used to detect potential program anomalies from most probable to least probable.

In [31], Murali et al. proposed a static analysis approach that extract inter-

procedural path-sensitive constraint repository by analyzing program source code. The constraint repository contains sequences or chain elements that identify the precedence relationships. To find frequent patterns in the trace-like chain repository of elements they performed a sequence mining operation. The resulting frequent patterns can be used to detect anomalies in the program.

In [16], Gabel et al. introduced a new technique based on Binary Decision Diagram (BDD) to mine small size finite state automata efficiently from the program execution trace. They represented their specification mining approach under a certain setting. For example, the technique requires a set of particular automata template. For each given particular template, check all traces of all possible concrete finite state automata that follow the template that are satisfied by the given traces. They used a BDD-based approach to speed up the process where they mined small size automata from traces of sizes up to millions of method calls. Moreover, in [15] they extended their mining technique by introducing an approach to merge multiple small size finite state automata to the larger Finite state machine.

In [40] Zhong et al. generated specifications by statically analyzing the program source code. The specification is in the form of a rule graph instead of Finite State Automata or sequence diagram. Each node and edge in the rule graph correspond to method calls and relationships between caller and callee methods respectively.

In the beginning, they started with a set of basic known rule graphs that can be generated from high-level knowledge about the program under inspection. Static analysis of the source code produces some new rules (e.g., facts) which are then combined with basic rule graph. Thus old rule graphs are extended to form new

bigger rule graphs in an interactive manner. In the end, all those rule graphs are visualized to show the relationships among various methods graphically.

3.1.2 Dynamic Analysis

Dynamic specification mining not only helps modelling behavior of a software system, but is also extremely useful for a wide range of software engineering tasks, such as requirement generation, validation and verification, anomaly detection [35], and test case generation [34]. The core of the dynamic specification mining lies in execution traces. They typically consist of sequences of method calls, and other related information. These sequences can be generated by profiling the program and running the system with different inputs (different scenarios), the more the better, to cover the overall behavior of the system. Hence producing correct and valid specification of the system. Higher coverage of the test inputs generates more accurate and complete specification models.

Instrumentation

Gathering traces from real program execution is known as "instrumentation" which is performed by adding extra code in different layers of an application. There are two main types of instrumentation. One, in which source code is modified directly to record debugging information at certain locations. Second, in which modification related to instrumentation is inserted directly into compiled binaries once they are loaded in the memory. The first type of instrumentation can be further divided into sub-types where the instrumentation related code is added either 1) in the source

code, 2) at compile time and 3) post link time.

Aspect Oriented Program or AOP, introduced in [20], is a relatively new technique for instrumentation. It belongs to the compile-time instrumentation and normally is applied where modifying large code bases to access the underlying properties and behavior of the program is in-feasible. The technique gives users a way to segregate the domain/design and debug-related code bases. For instance, several libraries, including log4net [24] and log4j require logging code to be written inline. Although logging and tracing are essentially two different forms of information gathering, their lines of code are generally not related to the domain of the code base it is included in.

AOP helps in removing the clutter and organizes the instrumentation-related code `outside the original code`. This also takes care of the redundant code added for logging or tracing the application. In theory AOP greatly helps in maintaining code bases especially in production environments, where development and operations teams can maintain their own repositories without polluting each others' code. AspectC++ [2] is the language extension of C++ to incorporate AOP with C and C++ code bases. AspectC++ has been tried as one of the options to instrument the code base AspectC++.

Popular Techniques

In [7], Briand et al. generated program models as UML sequence diagram from program execution traces. A UML sequence diagram represents objects lifecycle and activity. It also represents messages for each of the method calls between the

corresponding objects.

They instrumented a java distributed system application using AspectJ (Aspect-Oriented Language for Java) in a framework to collect execution traces and analyzed the produced execution trace to form sequence diagrams that emphasize on the method calls; showing caller, callee, and signature information. The produced sequence diagram represents branches, loops, and start and end of scenarios.

However, their abstraction procedure is rather high-level and may not be suitable for extracting in-depth specifications. For example, they mapped concrete numeric values n to only three abstract states e.g., n<0, n=0, n>0 which does not represent any business logic of the observed object. Thus, they mapped all concrete states to some abstract states. To summarize the behavior of a single class they merged all the object behavior to a single Finite State Machine (a state machine representation with states, and transitions that fire when certain event of condition trigger) that contains all object states and transitions.

A Finite State Machine (FSM) [37] is a model that is often used for depicting the high level behavior of a software system. Different forms of FSM are used for the said purpose[12]. It consists of a set of states, interconnected by labelled and directioned transitions. A State in FSM is a unique state of the system at any given point, which is caused by specific event called Transitions such that FSM looks like a flow in which program is changing its state from one to another on the application of transitions. Transitions in FSM are multi-directed; i.e. single transition can result in changing the state of system from State A to State B and also State B to State A.

Most of model inference techniques, generating Finite State Machines(FSM), begin

by building a Prefix Tree Acceptor (PTA); a tree shaped diagram of interconnected states showing the flow of a program in terms of method calls. PTA is generated from concrete execution traces, and contain the exact paths that are acceptable by the system. Since this step is a mere translation of traces to a tree, it contains a huge number of states and has recurring behavior all over (no abstraction).

In [10], Dalmeier et al. introduced a prototype called ADABU for JAVA programs that mines models after classifying all the methods into two categories: Mutator; that changes the state of an observable object and Inspector; that reveals properties of that object. Their mining approach dynamically captures the effect of mutator methods of the object state by calling all possible inspectors before and after calling each mutator. Therefore, the abstractions are based on the return values of inspectors. My approach is in line with ADABU, where for any function to be included as a transition, it has to be a `mutator` and all the `inspectors` are associated with states.

The other way of abstraction is through state merging which also reduces the number of states, still keeping the state machine concise as well. When a state B is merged with state A, all incoming transitions to B are pointed to A and all outgoing transitions of B are modified to leave from A. In this way recurring behavior is merged with in a state machine. Several techniques in the past have applied different ideas to merge the states. Successful inference of software behavior through a summarized yet concise model depends upon the validity of the state merging approach used. I have explained some of these approaches below.

[5]'s k-tails algorithm iteratively merges the pair of states in a state machine, if they are k-equivalent, until no such pair is remained. K-equivalency here denotes

that both states have same sequence of leaving transitions of length k. The state machines generated by the above technique depict the program behavior, and apply merging techniques on a sequence of function calls. However, data constraints are an important aspect of program behavior and if not included, the inferred program is deemed incomplete. In this regards, Extended Finite State Machines(EFSM) are helpful since they annotate the edges in FSM with data constraints valid for that interaction.

In [38] Walkinshaw et al. presented the QSM technique which used Prince's blue-fringe merging algorithm [22] to compute the scores of a state pair and merge the ones with positive score. It compares two states, incrementing the score for every overlapping transition label in the suffixes of both states. After merging the state pairs with positive scores, the technique poses questions to the end user whether the new paths in the state machines formed due to merging (if any) are valid. If valid it merges the states and repeats the process on the set of compatible states. However, the technique restricts the set of pairs that can be merged generalizing lesser number of states, and also produces a large number of questions for real world programs. It also becomes invalid if the system have complex data constraints that are affected by the inputs of transition functions.

In [26], Biermann's GK-Tail is another abstraction algorithm in which Lorenzi et. al. adapted the k-tails algorithm and extended finite state machines with information of data constraints observed during program interactions. GK-Tail merges similar states based on state invariants generated by Daikon [13]. Their technique used a 4 step procedure to generate an EFSM. It starts with merging the input

equivalent traces (sequence of same method calls but different parameter values), and then uses Daikon to generate predicates from data values in the traces to produce an initial EFSM with variable constraints. It then merges the equivalent states by comparing the outgoing transitions of the states augmented with data invariants after categorizing them in different categories.

Daikon's invariants can either be a range of values, or relation between different properties that are helpful in defining behavior of the program with respect to its properties. However, like other invariant detectors, Daikon operates at the level of functional granularity, hence it analyses the parameters that are passed to the functions.

Daikon is a tool for the dynamic detection of likely invariants. It infers invariants by observing the variable values computed over a certain number of program executions. They can either be single value invariants ($a == 1$), or relation between different properties ($x = y + z - 1$) that are helpful in defining behavior of the program with respect to its properties. However, like most invariant detectors, Daikon operates at the level of functional granularity, hence it analyses the parameters that are passed to the functions. In [23], Maoz et al. presented another usage of Daikon in which they generated invariants augmented Live Sequence Charts.

Moreover, in [21], Kim et al. evaluated daikon to report that for different implementation of a simple add function; one by calling `c=add(a,b)` and the other by calling `c=a+b`, daikon didn't generate predicates for variables that are not defined as a parameter of any function, or are not returned in a function. Kim et al. also stated that it could not handle large applications, which is mostly the case in real world

programs. Moreover, for automated invariant detections, a large number of test cases or user input is required to come up with useful constraints. Same is true for the other inference techniques in which a complete test suite is required to generate valid generalized behavior in order to hold true for any new interaction in the system.

3.2 Debugging

Traditional debugging techniques include logging, assertion, profiling, and breakpoints [39]. Logging is performed by printing the program state or general messages through code instrumentation. Assertions are constraints, that are added to desired locations in a program, that when become false break the program. Profiling is the run-time analysis of performance of the program by monitoring its memory, cpu, etc. usage, which can help in detecting bugs susch as memory leaks, unexpected calls to functions etc [39]. However, in practice breakpoints are the most common method for debugging, where the code is paused at the location of a breakpoint and the developer can inspect the variables and other context information at that instant of the execution. Effectiveness of breakpoints largely depends on the knowledge of the developer to put these breakpoints at relevant locations.

Another type of breakpoints is called conditional breakpoint, in which debugger evaluates the condition containing program variables at the location of breakpoint and only pauses the execution if it becomes true. However, they make the program extremely slow due to the reason that conditional breakpoints are not supported by hardware by default. Debugger processes such breakpoints as normal breakpoints and evaluates the condition to decide whether it should break of resume operation. More

advanced debugging techniques make use of execution traces to get insight from the execution of program, which can also be used for monitoring the system in run-time. This also depends on the level of information available during dynamic analysis to log in the traces.

Several development framework have logging capabilities which can be used by developers to log statements in certain events (warning, errors etc). However, the events are limited and require a lot of developer's effort to inspect the code looking at a log statement.

In [33], Spinellis considers execution traces better than application-level logging for several reasons, especially for the frameworks that lack default logging capabilities, including the ability to debug the faulty situations in production environment after a full execution of the program.

As mentioned above, putting debug breakpoint in a program further reduces the performance of the system, specially when debug conditions are used. This method is also not feasible in the case of execution of huge programs in which thousands and millions of interactions are made during a single run. Considering all the drawbacks, collecting traces during the real execution of program in production environment seems very practical and opens a wide array of possibilities to present the collected data in an organized way. Traces are collected by profilers that are hooked with the program and trigger on certain events of execution, but have similar access to program objects as the program. And since no change is made to the code, code access is also not required when collecting traces.

In [28], Maoz proposed the idea of using the abstracted traces as run-time models

when a program is run against a defined high-level model, collecting relevant low-level data during the execution of the program. Maoz also used the idea of selective tracing, hence limiting the scope of traces to the scope of the higher-level models. The presented technique in this paper also uses the same basic idea of limiting the scope of execution traces to improve the efficiency of the tool and also help developers to focus on the selected context.

Using the idea, in [36] Vogel et al. proposed mega-models at run-time, a technique to maintain run-time models of huge software systems, which help with model-driven environments, when there are multitude of models related to each other.

Another related category of studies is model-based debugging, where the main idea is automatically generating models from real execution of a program, and identify the location of the bugs by comparing its expected (given as models) and actual behavior (extracted) [4]. In [30], Mayer et al. also used the same idea and applied artificial intelligence techniques on run-time models to automatically report suspected faults or assumptions in the case of deviating behavior. Other applications of the idea in the same category are also applied in [14; 29]. In [32], Shang et al. used the same idea for debugging big data applications, but extracted models from Hadoop framework's log files rather than execution traces. However, fully automated techniques are not generally considered feasible for the real world programs due to a large number of reported false positives.

Other categories of related work that are slightly more remote and I do not explore in detail, are extracting models for program comprehension discussed in [9], anomaly detection based on dynamic analysis presented by [27], and the entire category of

fault localization, a survey of which is presented in [39].

Chapter 4

Methodology

The approach presented in this thesis gives users an interactive semi-automated technique to extract a focused specification of the system at the right level of details, in each context. The main idea of the approach is to let the users select the context of the abstracted state machine. The approach is divided into three steps: 1) Extracting execution traces, an interactive step to get the user input in terms of important functions and state fields. The selection helps in selective instrumentation of the code and get corresponding execution traces. 2) Abstraction and merging of multiple executions in a concise state machine. 3) Presenting the state machine in a user-friendly view. An overview of the proposed approach is shown in Fig 4.1, where the steps are explained in the subsections below. Since the approach is adapted to be used by an organization, it is bundled into a web application. The details of the web application will also be explained in this section.

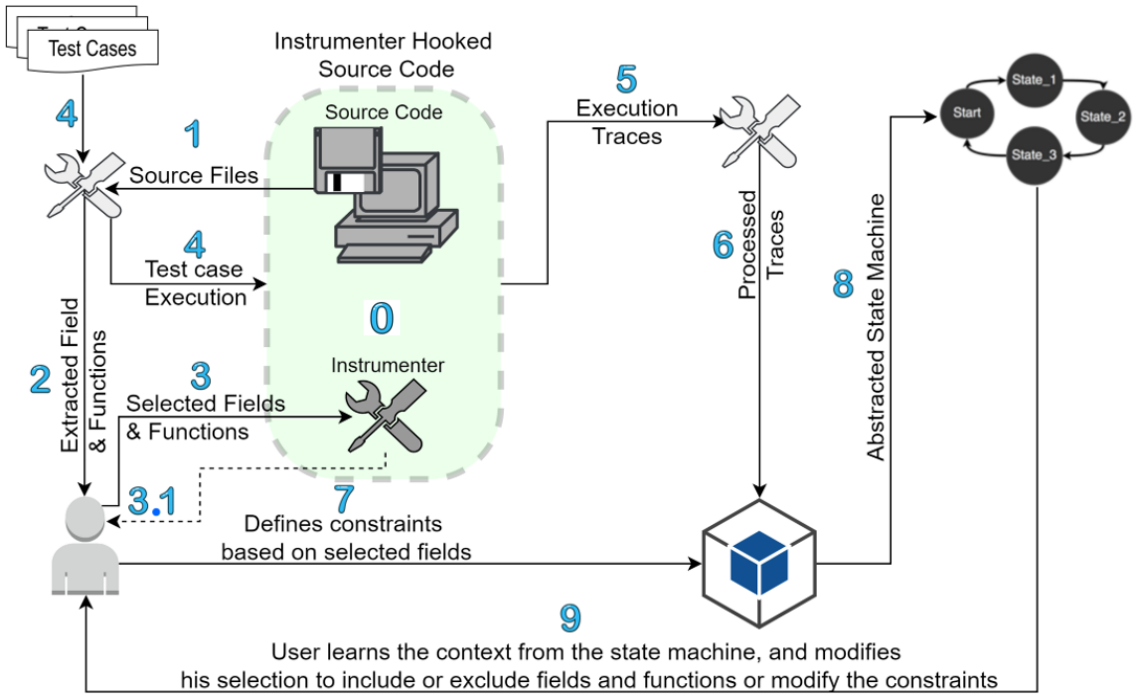


Figure 4.1: Overview of the proposed interactive specification mining approach

4.1 Generating execution traces

As mentioned in the background, one of the elegant ways of extracting execution traces is using Aspect Oriented Programming. Appendix A.1 shows more details on AOP and how we managed to apply that on small code bases. However, due to the limitations of the discussed tool, `aspectC++`, in handling complex makefiles and issues related to the usage of g++ in compiling both C and C++ files, this option was not feasible, in our context.

Since Micropilot's code base is maintained as several Microsoft Visual Studio solutions, our best alternative to AOP was using compiler options for tracing. In this case, the visual studio compiler options of /GH[1] and /Gh were used for profiling and instrumentation.

Therefore, in **step 0** of our approach (Figure 4.1), to enable the above options, the methods of *_penter* and *_pexit* are hooked to the entry and exit points of all functions without adding any additional lines of code to the original code-base. Inside the *_penter* and *_pexit* functions, additional code is added for writing the execution traces. The code uses the context saved in program registers and manipulates them to obtain references to functions and fields for tracing. It is implemented as a static library and its output file is linked to the subject code base. After compiling and linking process, the instrumentation-related code is inserted directly into the compiled binaries, when they are loaded in the memory. More details on the *_penter* and *_pexit* functions are explained in the appendix Entry and Exit Hook Functions.

The instrumentation step has an optional feature of recordings call stack trace. If the option is ON, one can see the entire `call stack trace` (all previous function calls) information by hovering over any transition in the state machine. To implement this feature, "Stack walking" is used with the help of`DebugHlp.lib` to get the names, line numbers and parameter values of the functions in the stack trace. Since this is an expensive operation due to the use of the library, the feature is optional.

The hook methods stated above are called on all functions' entry and exit events. However, tracing all functions in the code is highly inefficient and costly. Normally, such functions also include primitive functions like printf and getch. Hence, there is a need to limit the context of traces by limiting the functions and fields to log.

Therefore, in **step 1** of our approach, a static analysis tool is used to read the raw source files. It extracts a list of global fields and functions and is based on `Exuberant CTAGS` presented in [19]. For a given source file, ctags generates a list

of tags for all global variables and functions in the code. The tool generates tag files for each source file in the code-base. It compiles a list of state fields by first analyzing the fields and structs, and then recursively analyzing the contained properties if they are structs. It is highly configurable and can be used with any code base of languages c, c++, java and c#. It was first developed as a c# dynamic library but eventually the module was incorporated in the main tool to be executed by the web application on the run.

In **step 2**, the extracted lists are displayed in an interactive GUI with several filters. The user selects the context of the state machine by selecting the related fields and functions with respect to his requirement (e.g., debugging a given bug). Again, initially developed as a windows application, it was replaced by a web page in the main tool.

An important feature that was requested by Micropilot developers during the first round of interviews (See Section 5.7.1), was the automatic selection of both the list of functions that modify a selected field, and the list of fields that are modified inside a selected function. Clang C++ parser [[3]] is a well known tool that could help implementing this. However, the tool requires the code base to be compiled with the tool (outside visual studio), which is not an option with our code base.

Therefore, a custom mini-parser was developed that parses the files and maintains a list of all the functions and the fields modified inside them. While parsing a function, it records all the fields modified within the function. If there are any other function calls inside the function, it parses those functions and includes the fields modified by those functions in the same list. By manipulating the list, it also shows the list of

functions modifying a field if a field was selected. The lists are used to display the information as explained in the Select below.

After a selection is made by the user in **step 3**, references of functions and their names from the subject program are passed to the profiler (library containing definition of *_penter* and *_pexit* functions explained above). The code for this is generated and appended automatically to the main file of the subject code.

In **step 3** all the functions in the subject source code are analyzed and made available for selection regardless of their accessibility in the main file. At this point, verification of the selection is necessary through compilation. The reasons for this is that in practice many functions are declared as private by design. In C/C++ code bases, this is done by excluding their declarations from the header files. Since they are not accessible by the main function, their references can not be obtained and passed. The user is instructed to remove those functions from the selection in **step 3.1**. Once compiled successfully, the subject program can be executed to obtain execution traces.

In **step 4**, the tool executes the code-base with a list of given test cases or scenarios. Once the code-base is executed the traces are gathered in **step 5**.

The raw log files of selected traces collected in **step 5** can be quite large in real world programs, due to several function calls that do not affect the selected fields. To minimize the traces, two options were possible. 1-Only trace those functions that result in change of state (fields). 2- Trace all the selection functions and then process the trace file to keep only the functions that affect the selected fields. The former option is more expensive due to the frequent comparison of state fields in all functions

calls. Hence, the latter approach was chosen in **step 6**.

The change impact analysis of function calls on state fields is performed by tracing the state fields at the entry and exit points of the functions. By comparing the "before-enter" and "after-exit" values of the fields, it can be determined which function changed which fields. The comparison is simple for the functions which do not call any other function. If both values are different, then the effect is produced by this function. However, for the changes made inside nested calls, there are three possible scenarios; 1) The change is made by the original function before nested call, 2) The change is made inside nested function, and 3) The change is made by original function after nested call. For these scenarios, the information can be obtained by maintaining a stack of "before-enter" and "after-exit" values and comparing the respective values to determine which function made the change visible in the state fields. Recursive calls are made for the functions when the change is made by a nested call. The scenarios are explained further with examples in appendix Trace Processor.

As mentioned above, the main reason for processing trace files is reducing the trace file size. Which in my experience reached up to several GBs for single execution, if a large list of fields and functions are selected. Reducing the size of trace file also reduces the time taken by the abstractor, explained in the next section. The improvements had massive effects on the performance of the abstractor, reducing the turn around time for the developers for analyzing an issue.

4.2 Interactive Merging

The approach for state abstraction and merging proposed approach in this thesis is based on the idea of GK-tail. It differs in the way that instead of generating invariants for method parameters automatically by Daikon, the users define constraints on the selected fields (**step 7** of Fig 4.1) and the abstractor associates them directly with states. The constraints are defined using a template selected from list of constraint templates, which are designed with our industry partner's consultation, and can be extended for other application domains.

In practice, the user selects one or many constraint template(s), and define them using the state field(s). The concrete traces are then abstracted to only show states that can be uniquely identified using the combination of these constraints. The current set of templates are described in the Table 4.1. Each template is explained with an example based on the example code discussed in Section 2.

Template 1 (`ValueChange`): This constraint accepts one state field (X). Based on this constraint, a new state is generated when the system detects a change in the value of X. This is an important category as most of the state dependent systems (specially in the embedded software domain) keep track of internal states using several fields, where each integral value represents a unique state of the system. The corresponding example in Table 4.1 shows the states generated due to change in value of field `gear` (`gear==0` and `gear==1`).

Template 2 (`ComparedWith`): This constraint accepts two state fields (X and Y). Based on this constraint, a new state is generated when the system detects a change in the relationship between X and Y. For instance, if X and Y are `speed` and

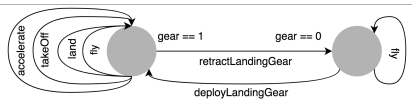
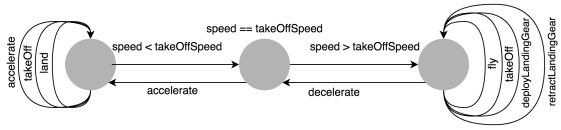
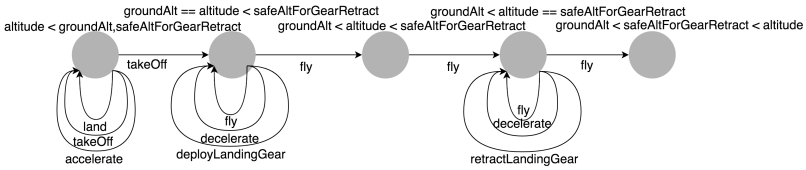
Category	**Example**
ValueChange of X	
X comparedWith Y or constant	
X comparedWith a Range (Y,Z) or (const,const)	

Table 4.1: Categories of Merging Constraints

`takeOffSpeed`, respectively, there will be three constraints generated as `speed` < `takeOffSpeed`, `speed` == `takeOffSpeed`, and `speed` > `takeOffSpeed`.

Template 3 (`ComparedWith a Range`): This constraint accepts three state fields (X, Y, and Z) and is an extension of previous template. Based on this constraint, a new state is generated when the system detects a value change of field X against the interval (Y, Z). Thus the possible states would be: X < Y, X == Y, Y < X < Z, X == Z, Z < X. In the example from Table 4.1, `altitude` is compared with [`groundAlt` , `safeAltForGearRetract`]. Note that, in general, Y in the previous template and Y and Z in this one can be system variables or constants.

The templates defined above categorize the relations between the selected fields in the the most basic ways which is normally sufficient. However, the templates are scale-able and new templates, targeting complex relationships between fields, can be

added during maintenance of the system.

As discussed above, the user can define one or multiple constraint(s). There can also be more than one constraint per field. To abstract state machine from a concrete trace, the tool reads the concrete execution traces sequentially. For each state it checks the defined constraints in OR fashion against the current set of values of the respected fields and determines whether a new state should be generated in the system. If any of the constraints have changed, a new state is generated. If all the constraints are the same as the last state generated then a new state is not generated and abstractor moves ahead. Hence, each selected function call in the trace connect the transition to A) the same state (loop), B) one of the existing states other than "self", or C) newly created state. For A, a loop is showed over the last generated state. For B, a transition is made to leave from the last generated state and linked to the existing state. For C, a new state is generated with the transition leaving from the last generated state.

Extra Template (`TraceStates only in Range`): Another important practical feature of my tool in the abstraction step is to setting a limit as an interval for a state field. The limit is defined the same as other constraints and is called `TraceStates only in Range`. This constraint is not an abstraction mechanism per se, but it helps the extracted state machines to be practical, specially, when using Template 1 in a debugging application. The constraint sets a limit on a field value X as a range and only considers parts of the trace for abstraction, where the X is in the given range. This is helpful in the cases when a frequently changing field needs to be selected under `ValueChange`.

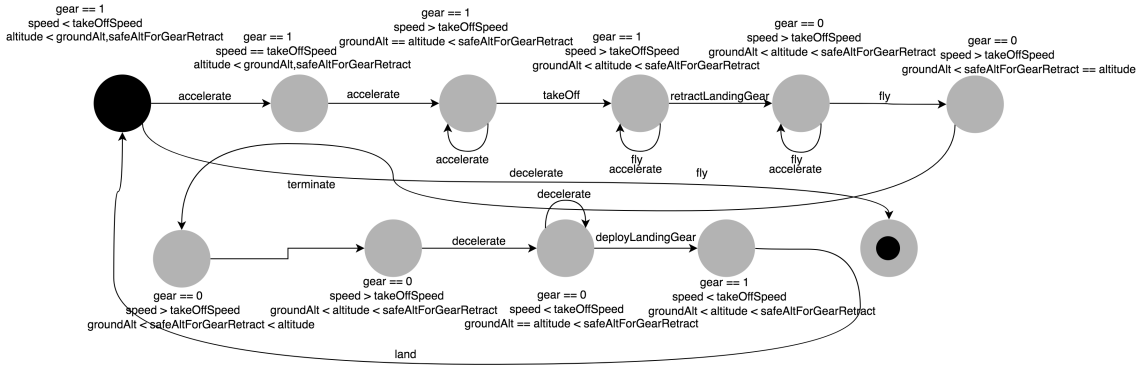


Figure 4.2: Example of final state machine with combined constraints

Finally, in **step 8**, all abstracted traces are combined into one state machine. In fact, it is not one step alone and the state machine is generated gradually when each concrete trace is abstracted and appended to the existing state machine (starting from nothing, to one path and finally becoming a full representation of all monitored traces). After all the abstracted traces are appended to the combined state machines, the states are iterated and all the states with the same set of constraints are merged. While merging two states S1 and S2, considering S2 the state to be dissolved, all the incoming transitions of S1 are now pointed to S1. Similarly, all the outgoing transitions from S2 are now modified to leave from S1. Eventually, I get the final abstracted state machine that depicts the combined behavior of the system from the selected trace files. A sample state machine with the combined constraints of Table 4.1 can be seen in Fig 4.2.

Extra Feature (`Composite States`): The last practical feature of my tool in the abstraction step is allowing "Composite States". Basically, whenever, I abstract concrete states into higher-level states, I do not discard the low-level states. The tool allows the user to select a state and zoom in. This lets the user to see all the concrete

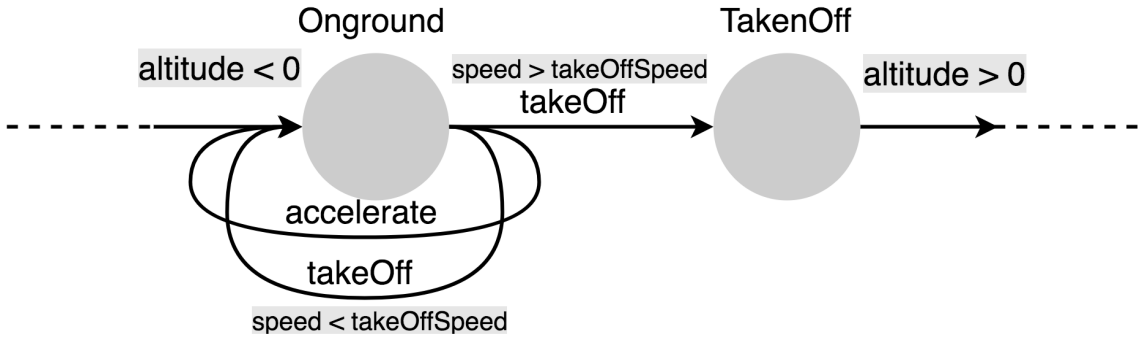


Figure 4.3: Guards on transitions

states and transitions (actual method calls) within an abstract state. This is the perfect debugging feature where the user can zoom into the problematic states and dig into the issue with all execution-level details without being lost with in a detailed full state machine.

Guards: In the case study, like most real-world embedded C programs, state fields are maintained globally. Thus, most functions are defined without any parameters or with objects of state fields rather than a list of variables. Hence the guards are constrained over state fields. Since constraint over state fields are already obtained from the user, the user is asked to differentiate between state constraints and transition constraints (guards). The guard constraints do not create new states. They only show the precondition for a transition and the destination state of the transition needs to be uniquely identified by other constraint(s). Fig 4.3, shows an example of guards that are generated in this manner.

4.3 State Machine Presentation

In **Step 9** of the Fig 4.1, the user is presented with the final abstracted state machine in an interactive web form, where the user is presented with the abstracted behavior of the executed scenarios. User can also modify his definition of constraints and start the process again from **Step 7** without performing another execution. Since the tool is developed as a web application, the abstracted behavior is presented to the user in a web form. An open source library known as JointJs is used for displaying state machines in an interactive way where users can move the states and transition around if any information is hidden (in case of a large number of states). There are several other features that are available to the user to interact with the state machine and also to view all the information in a convenient way. The features are explained in the section below.

4.3.1 Web Application

The tool is developed as a web application due to several reasons. The main reason is supporting concurrent user, without having to maintain several versions. It also gives a relatively secure access to the tool from outside company. Finally, it is platform-independent and provides a user-friendly GUI.

The remaining sections of this chapter explain the individual features of the tool categorized by the "views", as follows: 1) Select, 2) Define and 3) Generate; where all the business logic is maintained at the "business layer", in the back-end.

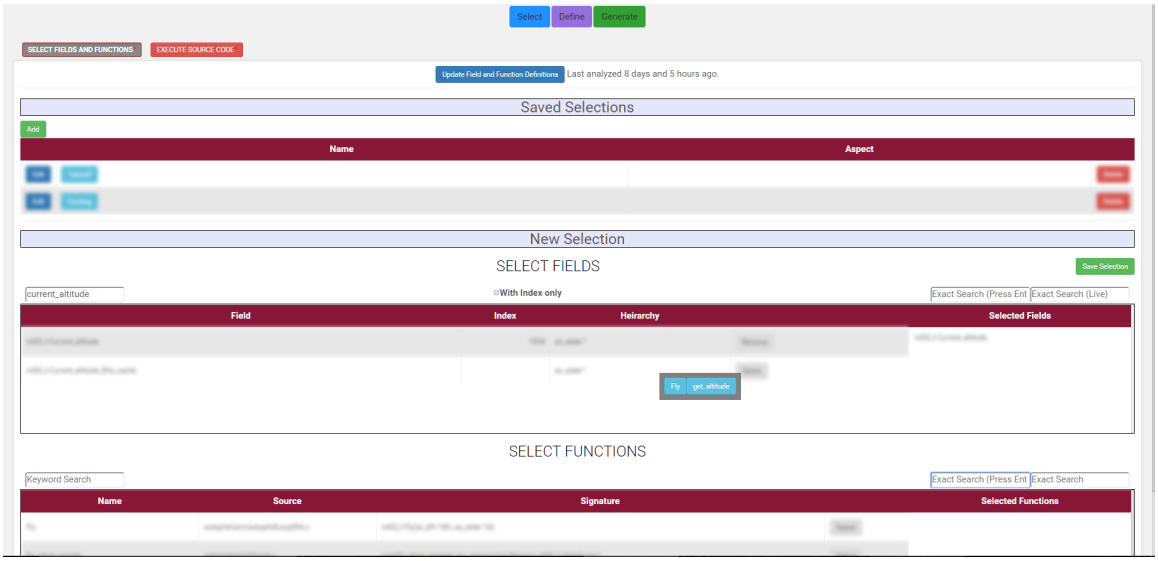


Figure 4.4: Selection - Select view of the Tool

Select

This view lets the users run the static analysis tool explained in Generating execution traces. The analysis is divided in four steps. 1) Header files analysis for state fields extraction, 2) Source files analysis for functions extraction, 3) Using the analysis from step 2 and reading the respective source files to get the information of starting and ending line numbers for all functions and 4) Parse the source files to extract the information of modified fields inside the functions. The operation is slightly expensive, since it logs everything (all fields and functions). Hence the result from each analysis is stored in a database and loaded when the view is opened. However, now that this data is recorded in DB, any user can get instant access to the results without re-running the analyzer. In other word, the expensive execution needs to be repeated only if the code has changed. As stated above, the view also presents the user two search-able lists per fields and functions. The lists support keyword search and can

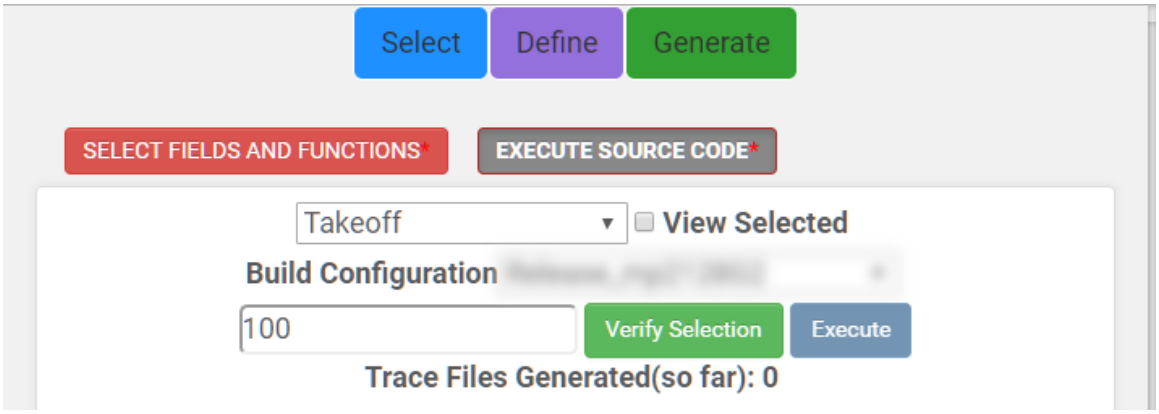


Figure 4.5: Execution - Select view of the Tool

also be searched by exact field/function names, file names (where the fields/functions are defined), by struct names (where the fields are defined), or by the field index (a standard filed ID used at the company).

By hovering the cursor over any field in its list, a pointer-following tooltip displays a list of functions the field is modified in. Similarly, hovering pointer over any function displays a tooltip displaying all the fields the function modifies. Once any field/function is selected this way, its corresponding list will be updated. The entire list of selected fields and functions can be saved as SELECTION. A screen-shot of the tab explained above can be seen in Selection - Select view of the Tool.

In another tab, the user is displayed a drop-down list of SELECTIONs, a drop-down list of "build configurations" of the subject source code, and a text-box for "test scenario" or "case id". After selecting one item from each option, the user can press the "verify" button which compiles the current selection with code base. If the compilation fails, the errors are shown in a dialog box which asks the user to remove certain functions from the selection and try it again. If it succeeds, user can press the execute button which executes the source code at the server and displays a list

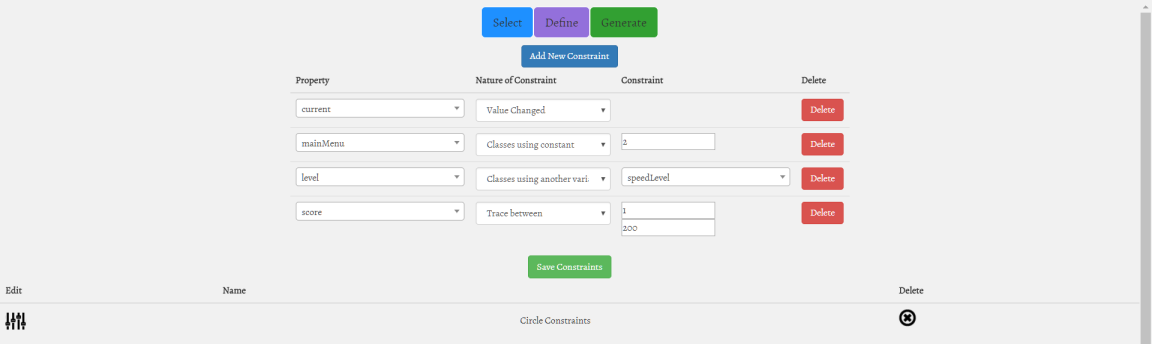


Figure 4.6: Generate view of the Tool

of trace files generated during the current execution. The status of the trace files is updated every two seconds showing their current file sizes and whether they are ready after a finished execution. Once the trace file(s) is ready, user can move to the next view to define constraints. A screen-shot of the tab can be seen in Execution - Select view of the Tool. Names of fields, functions and build configurations are hidden for confidentiality reason.

Define

This view is to define constraints. On the view load event, it reads all the trace files in the configured directory and extracts all the state fields. The users can select a field from the list and a constraint from the predefined list of constraint templates. They can can add as many constraints in one definition and save it for later reuse. A screen-shot of the view is shown in Generate view of the Tool.

Generate

The view uses the data saved in the previous views to generate a state machine from the selected execution trace(s). In a tabbed form view, user is first asked to select

the trace file(s) from a list which shows the execution time, platform, dll name and file size for each trace file found in the configured directory. On selection, only valid definitions (of the selected trace files) are loaded in a drop-down menu. A definition is considered valid only if all the properties assigned to its different constraints exist in the selected trace files. The user can click on generate button after selecting a definition, which generates and displays the state machine.

As discussed earlier, there are several extra features on top of the basic state machine presentation, which are briefly explained here: a) displaying constraints (when hovering the pointer over states, a pointer-following tool-tip shows the respective constraints for each state), b) effects of transition (when hovering over a transition, the state values of the left and right states of the transitions are compared and difference is reported as the effect of this transitions), and c) the actual values of state fields (when clicking on a state, all the state fields are shown with their current values, converted to their units, in the side panel of the view).

Another feature of the tool is to see the possible reasons for a transition. For big programs in which state fields are maintained globally, most functions are defined without any parameters or with struct objects of state fields rather than a list of variables. Hence extracting guards over transitions from function parameters is not an option. For this purpose, the same list of selected fields is used as explained in the first subsection. All the fields in the selected trace, that are not included in any of the constraints, are monitored. Moreover, call stack trace of the function is displayed on hovering pointer over any of the transitions.

The tool is proposed to be used for debugging. Hence when showing an abstracted

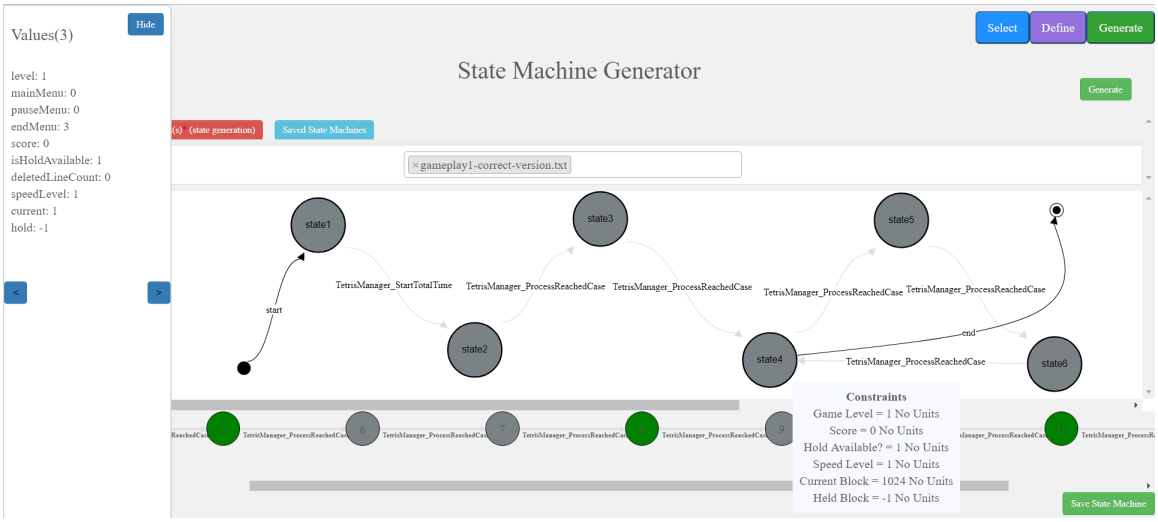


Figure 4.7: Generate view of the Tool

state machine, a corresponding un-abstracted version of the trace is also shown below the abstracted state machine. This figure is a mere translation of the trace file. It shows a straight flow of the program with the un-merged states linked with the merged states of the abstracted state machine. When clicking a state in the abstracted state machine, all the states that are merged into this state are highlighted in the un-abstracted state machine. On clicking the states in this level, the actual values of the fields are shown.

The abstracted state machine generated in this view is also highly customize-able. Managers can use this feature to propose new changes to existing program behavior by adding new states, adding more transitions from and to existing states, moving the states all over the view, adjusting the arrows of transitions and editing the names of transitions etc, and can save after. Saved state machines are showed in a new tab on this view, clicking on which opens the saved state machine in a new view. The Generate view is shown in Fig 4.7.

Chapter 5

Empirical Study

5.1 Objective of the Study

The goal of this study is to investigate the feasibility and effectiveness of my interactive specification mining approach for debugging, in a real-world industrial setting, and also the usability of the tool from a developers point of view. To achieve this goal, I have broken it down to four research questions, as follows:

RQ1) Does our interactive specification mining approach correctly extract the behavior of the running system?

This question verifies that our approach, in general, is sound. Since there is no correct/reference state machine already in place in our context, the outputs are verified using developers knowledge.

RQ2) Is selecting the relevant fields, functions, and constraints, interactively by the developers, feasible?

This is the core assumption of our approach that the semi-automation is feasible,

in this context. Unlike most existing related work in the domain, I do not try to fully automate the abstraction process and get the information about the aspect and level of detail of the specification from domain experts. Therefore, the question is whether the domain experts are able to provide those inputs, in practice?

RQ3) Does the abstracted behaviour provides any extra useful insight compared to the current state of practice, for debugging?

Assuming the approach is sound (RQ1) and feasible (RQ2), the next question is whether it provide any extra insight for debugging that the current state of practice is missing.

RQ4) How easy is to adopt this approach in practice?

Finally, in RQ4, I briefly investigate the applicability and usability of the approach, in practice.

5.2 Interview

To answer the research questions, a qualitative study was performed by interviewing actual developers employed by our industrial partner. A total of three rounds of experiments were conducted at the company premises. During which, the developers were allowed access to the source code, a set of test cases, and issues. The test cases and issues were selected from their bug tracking system. All the issues used were encountered in the history of the system since a year before the experiment. The experiment, interview questionnaire and the issues are explained in detail in the following sub-sections.

5.3 Context

The study is conducted on a large code-base of a safety critical embedded system (an autopilot software for UAVs), owned by Micropilot Incorporation, the world leader in professional autopilots for UAVs and MAVs. The project used for the user study is an Autopilot System, that is a huge code-base with over 1.3 MLOC in hundreds of C/C++ source files.

The company is interested in acquiring safety certification (DO-178C) for its Autopilot software. One of the main requirements is to have explicit specification of the system, with trace-ability to source code and test cases, and vice-versa. The company is also interested in providing a better tool support for monitoring and debugging.

5.4 Subjects of Study

A total of eight developers (subjects) were interviewed in this study. Before the actual interview, the demographics collected from the developer is summarized as follows.

Six out of eight interviewees are software developers, one team leader and one working as a control engineer. One of the developers has a Ph.D., three of them have Masters degree in Computer Science, while the remaining developers have Bachelors degree in either Computer Science or Electrical and Computer Engineering. The total experience of six out of eight developers in the software industry ranged from one to three years. One developer had five years of experience while the team lead had 13 years of industrial experience, all with the same company. The subjects' level

of familiarity with the software under study (Autopilot) ranged from one to three out of five (self-evaluated), with the exception of the team leader who had a familiarity level of five.

A total number of five developers were included in the first round of the experiment. The second round consisted of two developers who had also been the part of the first round. The third and final round of the experiment was targeted towards the use of actual tool and three additional developers were used; one of them was a Ph.D. and the other was the team leader, with the most experience with the code base.

Both developers included in the second round were also participated in the first round, the second round was performed 4 months after the first round. Although the developers had seen the fields, functions and constraints for the issues, it is highly unlikely that they retained the information.

5.5 Pre-interview Tutorial

Three weeks before the first round of experiments, a short survey of the developers' knowledge on UML, State machines, and software modeling was completed, where they were asked whether they were familiar with those concepts. Six out of eight subjects were familiar with the state machines while two had seldom seen them during their academic studies. To refresh the background and making sure they are all at the same level of understanding with state machines in general, a brief tutorial was presented with the demonstration of the tool.

The tutorial was designed and run by the author and one industry staff (a senior

developer who helped the author in the design phase of the study and verifying the outputs, but was not among the subject group). During the tutorial, the subjects were demonstrated the process of generation of a few example state machines depicting the general behavior of the autopilot during several aspects of flight (e.g., takeoff, landing, etc). While showing them the state machines, the functionality of the tool was also explained. A very brief (5-10 min) training of how to read the output state machines followed the demonstration. The training focused on viewing information related to certain parts of the execution, and relate the flow of the state machine to user's knowledge of the system. They were also asked if they think the behavior of Autopilot shown in the sample state machines looks correct or not, where all the subjects verified the correctness.

5.6 Issues

For the actual interview, 5 real reported issues were used for debugging. Due to confidentiality reasons, the issues can not be revealed. However, they were selected with the help of the senior developer (not among subjects), to be representative of the company's typical issues.

The selected issues were already resolved. Therefore, they were reintroduced in multiple clones of the code-base, making 5 buggy versions of the code, where each version has only one known unresolved issue. All the buggy versions were validated by the lead developer as having the issues as taken from the bug tracking system.

The main criteria for selecting an issue were a) the bug should be realistically reproducible in the most recent stable version of the code (e.g., the older versions

of the code were not executable because of licensing changes of internal tool sets), b) the subjects must not be involved in resolving the issue, and c) they should be representative of typical issues encountered in everyday life of the developers in the company.

Points (a) and (c) were validated by the lead developer and point (b) was first checked against the history of code changes and involvement of the subjects on the selected issues and fixes. It was also validated by the senior developer.

The standard practice at the company after resolving an issue is to write a test case and include it in their regression tester, which can execute test cases on the code-base to verify the resolution of the issue. Hence the issue tracking system also contained information about the test cases that would cover the buggy behaviour and the actual assertions (which would not be known before localizing the defect). However, to emulate a normal debug scenario, the abstraction tool does not use that data.

5.7 Interview Design and Setup

As mentioned in the Interview a total of three rounds of experiments were conducted; second round 1 month apart from the first round and third round 2 weeks apart from the second round. The purpose of Round 1 was to get the feedback on the approach explained in Chapter 4. Round 2 was performed as a followup round for Round 1 where feedback for the newly added features (requested in round 1) were obtained. The last Round 3 of experiments and interviews was performed so that the users can use the actual tool with real scenarios and provide their feedback.

5.7.1 Round 1

The first round of experiments was designed to assess the ability of our tool to abstract the behaviour of a buggy scenario to help for debugging. The subjects were provided with handouts containing description of five issues/bugs from the issue tracking system. As discussed, the descriptions did not reveal the fix. Each issue came with its corresponding faulty and correct state machines representing the code after the bug fix.

In an ideal situation, the subjects would have been asked to use the tool from scratch (i.e., in an interactive way to find the best set of fields, functions, and constraints) and debug the code with the help of the insights given by the abstracted state machines. Then this process could be compared with their current debugging practice in terms of effectiveness and efficiency. However, the setup was deemed expensive by the company. According to the contact developer, debugging an issue might take several hours if not days. Therefore, the company could not afford giving required resources to the research team for such an extensive experiment for several issues.

Hence, an alternative experiment was designed which still answers the RQs to some extent. In this setup the subjects were allowed a limited time of maximum 10 minutes per issue. At the end, 10 minutes (plus any leftover from issues times) were assigned for answering the questionnaire, in a one-to-one interview. The interview was held at the company premises. The interview time including the experiment was fixed to one hour per subject. The sessions were recorded and transcribed later.

Given the 10-minute time per issue the core objective of our study was focused,

which was the feasibility and applicability of the approach. Thus, the subjects were not asked to fix the buggy code. To save the time, the relevant fields and functions were selected in the first round with the help of our contact developer, beforehand. Only the final state machines were provided to the subjects.

Since the selection step was already completed on their behalf. To validate this feature of our approach, the subjects were provided the list of selected fields and functions in the handouts. They were also asked if they would have selected the same fields for the issues.

As already mentioned above, there are two state machines given to the subjects per issue. These state machines are generated using the execution traces we get when we run the test cases that are added after the fix, per issue. Running the actual test cases (containing the test assertions) on the buggy code would result in a test failure and would reveal the bug. Thus in the experiment, the test cases were executed without any assertion so that the only differences in behaviour of the buggy vs. correct state machine are the effects of the bug.

In each session, the subjects were asked to (a) check if they can relate the faulty and correct behavior in the state machines to the given issue descriptions, (b) explain the differences in behavior in the two state machines, (c) identify the fields relevant to the issue, (d) locate the buggy function, and (e) answer some feedback questions about feasibility of the overall approach, effectiveness of state machines in finding bugs in general and usability of the tool.

5.7.2 Round 2

The second round of experiments and interviews was a follow-up to the first one. It was performed to get feedback on the newly added features that were asked by the developers in round 1. The two developers from the previous round that asked for the features were asked to take part in this round in which they were asked to use the actual tool to debug two issues. Due to time constraints the number of issues were limited to two. For the first issue, developers were provided an already selected set of fields and functions, and constraints. Hence, they just had to generate state machines. A total of three versions were maintained for (one) correct and (two) buggy code bases. Hence for one issue, they were required to use the tool twice to get buggy and correct state machine and inspect them as the last round of the experiment.

For the second issue they had to select everything by themselves. The second issue of round 2 and the entire round 3 were carried out to show the cost/benefit and feasibility of our approach by asking the interviewees to select the inputs and constraints. Another point was that since the developers were provided both the buggy and correct state machines, in round 1 and round 2 (issue 1), the given task was just a matter of spotting the difference between the two state machines. To provide a more realistic debugging scenario, in round 2 (issue 2) and round 3, the developers were asked to identify the location of the bug, without the correct version.

5.7.3 Round 3

The third round of experiments involved two mid-level developers and a team leader who had not seen the tool before the experiments. For this round, a total of

five buggy versions of the code base were prepared. Developers were asked to choose any two of them (after confirming from the bug tracking system that none of the developers were involved in the resolution of any of the five bugs). They were asked to find the location of the bug for one issue by their standard practice of debugging at the company. For the other issue, they were asked to use the tool and find the location of the bug, in terms of functions. For each bug, the allotted time was kept half an hour.

After the first half an hour, when the manual debugging period finished, the author provided a very brief demonstration of the tool, similar to what was demonstrated in round 1. During this demo the usage of the tool using a general scenario of Takeoff aspect of the flight was explained. In addition, the tool was demoed for one issue from the three that were not selected by the developer.

In the final half an hour, the subjects were asked to use the tool by themselves for debugging the selected issue.

During the sessions, the author observed their manual debugging practice and categorized it under the Bug Diagnostic Strategies explained in [6]. In addition, the number of executions performed by the interviewee, manually or by the tool, was recorded. Finally, the time taken by each execution during each session was recorded. At the end of interview, they were asked general feedback questions regarding the tool and the approach. The results from the experiments are reported in the section below.

5.8 Questions

All three rounds of interviews were designed in collaboration with the company team (including a high-level manager). Such that not only they answer the thesis's research questions but also assess the relevant questions to the company's needs and does not violate the employees rights and privacy.

For the first round, the final questionnaire includes three types of questions: a) Demographic questions (included in all rounds of interviews), where their answers are already discussed in the Subject of Study sub-section, b) Questions specific to debugging an issue, and c) Questions about the overall idea of the tool and general feedback on its feasibility and usability.

The lists of above questions are provided in Demographic Questions, Round 1 - Issue Specific Questions and Round 1 - General Feedback Questions.

Q1.1.1	Please state your total industrial experience.
Q1.1.2	Please rate your familiarity with the autopilot code base out of five.
Q1.1.3	What is your current position at micropilot.
Q1.1.4	What is your highest level of education?
Q1.1.5	Are you familiar with state machines in general?

Table 5.1: Demographic Questions

Q1.2.1	Explain the differences that you notice between the buggy and correct state machines (e.g.,number of states, sequence of functions, fields changes, etc.)
Q1.2.2	After comparing both state machines, are you able to identify the field(s) that caused the issue or showed the effect of the issue? After comparing both state machines, are you able to identify the field(s) that caused the issue or showed the effect of the issue?
Q1.2.3	Are you able to identify which function caused the issue?

Table 5.2: Round 1 - Issue Specific Questions

Q1.3.1	After looking at the list of functions and fields selected to a given issue, how difficult do you think it would be if you had to select those inputs for a given issue?
Q1.3.2	With respect to the selected fields and functions, are the state machines generated by the system correct? Are they detailed enough? If not, what important information is missing?
Q1.3.3	For given issues, are the set of selected fields and functions adequate? If you were selecting the fields by yourself, would you have made any changes to the list of selected fields and functions?
Q1.3.4	What is your current practice/procedure of debugging the system, when an issue is reported and assigned to you? Which tool support you have?
Q1.3.5	Assume you select the best set of fields and functions for a certain issue and use our tool to generate the corresponding state machines. Do you think our tool would bring any advantage to your current set of debugging/monitoring tools, at Micropilot Inc.? If so what advantages and if not why?
Q1.3.6	How easy it is to embed this tool into your current frameworks and infrastructure? Any challenges that you see in making this a part of your tool set?
Q1.3.7	Please provide your feedback about the feasibility, effectiveness, and usability of this tool and idea.

Table 5.3: Round 1 - General Feedback Questions

For the second round, the questionnaire included a couple of questions regarding the new features added; whether they added value to the tool and reduced the challenges in selecting a set of fields and functions for any generic issue. The questions are provided in Round 2 - General Feedback Questions.

Q2.1	How difficult was it to select the functions / fields, on a scale of one to ten, for a given issue?
Q2.2	After selecting the list of fields and functions, on a scale of one to ten, how difficult was it to come up with the valid constraints with respect to the scenario?
Q2.3	Do you think the newly added feature in the tool which displays a list of modified fields in a function automatically when you select that function (similarly, a list of functions modifying the field automatically load when you select the field) is advantageous over manually selecting both based on your knowledge?
Q2.4	How many times, on an average, did you have to repeat the process (select the fields, functions select constraints generate state machine)to abstract program behavior according to your need and expectation?
Q2.5	How much time did each go take on an average?

Table 5.4: Round 2 - General Feedback Questions

For the final round, in addition to the observations made during the sessions, the questionnaire only included question related to the feedback for the tool. The observations made are provided in Round 3 - Observational Questions, while the questionnaire for this round is provided in Round 3 - Feedback Questions.

Q3.1.1	How many times, on an average, did the developers have to repeat the process (select the fields, functions, select constraints and generate state machine) to abstract program behavior according to their expectation?
Q3.1.2	How much time did the manual debugging take, and how much time was taken by each tool run, average if multiple?
Q3.1.3	Which diagnostic strategies were used?

Table 5.5: Round 3 - Observational Questions

Q3.2.1	How difficult was it to select the functions / fields in general on a scale of 1 to 10?
Q3.2.2	After selecting the list of fields and functions, on a scale of 1 to 10 how difficult was it to come up with the valid constraints with respect to the scenario?
Q3.2.3	What is your current practice/procedure of debugging the system or any other tool that is handy, when an issue is reported and assigned to you? Which tool support do you have?
Q3.2.4	How beneficial is using this tool compared to your current practice?
Q3.2.5	How easy is it to embed this tool into your current frameworks and infrastructure? Any challenges that you see in making this a part of your tool-set? Would you use it.
Q3.2.6	Please provide your feedback about the feasibility, effectiveness, and usability of this tool and the idea of using your knowledge to break theissues down to fields and functions.

Table 5.6: Round 3 - Feedback Questions

5.9 Interview Results

In this section, I will provide a summary of the answers to all questions by the eight subjects of the interviews. The questions in this section are indexed as in Demographic Questions, Round 1 - Issue Specific Questions, Round 1 - General Feedback Questions, Round 2 - General Feedback Questions, Round 3 - Observational Questions and Round 3 - Feedback Questions. I will conclude by detailing the responses to my four research questions on the basis of the interview results.

5.9.1 Round 1

Q1.2.1: *Explain the differences that you notice between the buggy and correct statemachines (e.g.,number of states, sequence of functions, fields changes, etc.)*

Answer: Three out of five subjects were easily able to relate the descriptions in the issue to the behaviour illustrated in the state machines, without my help. They also stated that the behavior in the correct state machines conforms with the normal behavior of the program according to their understanding. However, two subjects were initially guided through the execution flow in the state machine, for the first issue. However, they managed to analyze the remaining issues without any help.

Q1.2.2: *After comparing both state machines, are you able to identify the field(s) that caused the issue or showed the effect of the issue?After comparing both state machines, are you able to identify the field(s) that caused the issue or showed the effect of the issue?*

Answer: All subjects were able to identify the fields that were affected by each issue, from the list of all selected fields in the state machine. The purpose of this

ID	**Subject 1** (Round 1)	**Subject 2** (Round 1)	**Subject 3** (Round 1,2)
Industry Experience (yrs)	2	0.9	2.5
Code Base Experience (/5)	3	2	2
Q1.2.1	all diff. spotted	all diff. spotted	all diff. spotted
Q1.2.2	5/5 correct fields	4/5 correct fields	4/5 correct fields
Q1.2.3	5/5 correct func.	4/5 correct func.	4/5 correct func.
Q1.3.1 (/5)	2	2	4 if not familiar
Q1.3.2	Correct	Correct	Correct
Q1.3.3	Adequate	Adequate	Would add more
Q1.3.4	Reproduce, debug with Visual Studio	Reproduce, simulate, debug with Visual -Studio	Reproduce, debug with Visual Studio
Q1.3.5	Visual component	Breaking debug activity to functions to get starting point	Big picture, suggests the functions we should review
Q1.3.6	Seems usable, no major problems	Makes debugging faster, worthy to get used to	Selecting fields is easier, coming up with constraints is difficult, not hard to get used to
Q1.3.7	F=9, E=8, U=8	F=7, E=7, U=7	F=9, E=9, U=9
Q2.1 (out of 10)	N/A	N/A	2
Q2.2 (out of 10)	N/A	N/A	2
Q2.3	N/A	N/A	extremely advantageous
Q2.4	N/A	N/A	One
Q2.5	N/A	N/A	$<$ 10 min

Table 5.7: Summarized responses to interview questions. F: Feasibility, E: Effectiveness, U:Usability

question is to verify if the subjects would be able to filter the fields, from a bigger list of selected fields, relevant to each issue.

Q1.2.3: *Are you able to identify which function caused the issue?*

ID	**Subject 4** (Round 1,2)	**Subject 5** (Round 1)	**Mapped to** (RQ)
Industry Experience (yrs)	0.9	1.5	N/A
Code Base Experience (/5)	2	2	N/A
Q1.2.1	all diff. spotted	all diff. spotted	RQ1
Q1.2.2	5/5 correct fields	5/5 correct fields	RQ3
Q1.2.3	5/5 correct func.	5/5 correct func.	RQ3
Q1.3.1 (/5)	2	2	RQ2
Q1.3.2	Missing details (branching inside methods, conditions etc)	Correct	RQ1
Q1.3.3	Would add more	Would add more	RQ2
Q1.3.4	Reproduce, debug with Visual Studio	Reproduce, debug with Visual Studio	RQ3
Q1.3.5	Visual component, suggests the functions we should review	Somewhat advanta--geous, doesn't state the line of issue	RQ3
Q1.3.6	Easy to learn and review	No challenges	RQ4
Q1.3.7	F=9, E=6, U=9	F=7, E=9, U=8	RQ2, RQ3, RQ4
Q2.1 (out of 10)	2	N/A	RQ2
Q2.2 (out of 10)	2	N/A	RQ2
Q2.3	extremely advantageous	N/A	RQ3
Q2.4	One	N/A	N/A
Q2.5	$<$ 10 min	N/A	N/A

Table 5.8: Summarized responses to interview questions. F: Feasibility, E: Effectiveness, U:Usability

Answer: Since the subjects were able to identify the the differences in behavior showed in the state machines, they were able to identify the functions causing abnormal behavior. However, the actual function causing the issue in 2 state machines were not selected, intentionally. For those state machines, three out of five subjects

stated that they don't think the function causing the error is included in the state machines. However, they correctly stated that the issue causing functions were called inside one of the selected function.

Q1.3.1: *After looking at the list of functions and fields selected to a given issue, how difficult do you think it would be if you had to select those inputs for a given issue?*

Answer: Four out of five subjects graded the difficulty of the process of selecting the relevant fields and functions with respect to any given issues as two out of five; one being the easiest and five difficult, whereas one subject commented that if you are not familiar with the code-base, it will be difficult to select the exact function but a keyword search can give an idea of where to start. However, the subject also commented that it is even more difficult to use Visual Studio for debugging the code if you are not familiar with the code-base, and it becomes a guessing game. Another interesting suggestion made by the same subject was to prepare a subset of fields and functions for every aspect of flight (or any other operation) such that if users want to debug an issue related to, say "landing", they would just have to select the aspect rather than a huge list of fields and functions.

Q1.3.2: *With respect to the selected fields and functions, are the state machines generated by the system correct? Are they detailed enough? If not, what important information is missing?*

Answer: Answering the question, three subjects said that they think the state machines were complete according to their understanding of the system, while two commented that they look complete but they missed some useful states that are to

do with functions that are not selected in the first place. Similarly, another subject commented that selecting fields were not as difficult as selecting functions, where you can miss important details, if important functions are missing in the state machine. In other words, the state machines generation is correct from their perspective but selecting an appropriate set of functions may be difficult in the first try.

A suggestion was also made to include all the important functions, which is impractical due to performance degradation and very large output state machines. One interesting suggestion was to warn the user if the system notices a change in the fields without any transitions (selected function calls) between them so that the user can analyze the change and select the relevant functions and repeat the step. Another suggestion was adding a new feature to perform more complex operators on the fields within the state machine, e.g., RADIUS(field1)>field2, which can be part of my future work.

Q1.3.3: *For given issues, are the set of selected fields and functions adequate? If you were selecting the fields by yourself, would you have made any changes to the list of selected fields and functions?*

Answer: Three subjects stated that they would have selected additional fields and functions for some of the issues and would omit the irrelevant fields. Two subjects thought that the selected fields were adequate for them to find the issue. But all of the subjects were able to identify the issue from the state machines. Therefore, we can summarize that even when all the relevant fields are not selected, subjects can still get an idea where something is getting wrong. In addition, since the process is interactive, users can adjust their selection and try different inputs and see the

results, immediately.

Q1.3.4: *What is your current practice/procedure of debugging the system, when an issue is reported and assigned to you? Which tool support you have?*

Answer: Answering the question, most interviewees responded that when an issue is assigned to them, they first try to reproduce the issue, as explained in the description, and run it on the simulator in their in-house tool. Then they move to Visual Studio, go through the code and use breakpoints to break the executions in different functions, then they step through the code while inspecting the values of certain fields.

Q1.3.5: *Assume you select the best set of fields and functions for a certain issue and use our tool to generate the corresponding state machines. Do you think our tool would bring any advantage to your current set of debugging/monitoring tools, at Micropilot Inc.? If so what advantages and if not why?*

Answer: Answering the question, three out of five subjects said the "visual representation" of the code made them see the big picture of the execution, they were able to see where code was branching and they learned something. In addition, it provided them an organized way to inspect the fields during the execution of the whole system and gave an idea on which functions to review and in which level of detail.

Two subjects also added that even incomplete state machines (with missing fields and functions or wrong level of details) are helpful in that they would have given them a starting point to debug (finding which functions to put breakpoints at). In addition, they mentioned that the stack traces that are accessible in the detail (zoom-in) views

are also helpful for adding the caller functions into the selected function list.

One subject also added that it is easier to explain the functionality of the system with respect to fields and function calls on the state machine compared to explaining the source code (i.e., program comprehension application of abstraction).

Another subject state that "if a relevant field is changing too many times, its hard to debug the function that is changing it. You might miss the important changes that you are actually interested in when skipping breakpoints. So, it's interesting to see how the field changes throughout the execution in the state machine and then go back to the code to inspect in relevant intervals."

Q1.3.6: *How easy it is to embed this tool into your current frameworks and infrastructure? Any challenges that you see in making this a part of your tool set?*

Answer: Answering the question, all subjects agreed that my tool would be a valuable addition to their tool-set and will be worthy to adapt to the tool. Similarly, all subjects thought that the idea was easy enough to be applied and thus is a valuable addition.

Three subjects said that if the tool was included in their tool-set, they would adapt easily to it. While two of them mentioned that any new tool is hard to adapt, in the beginning. However, once they get used to it, it would not be hard to use.

Q1.3.7: *Please provide your feedback about the feasibility, effectiveness, and usability of this tool and idea.*

Answer: In the final question, the subjects graded (out of 10) feasibility, effectiveness, and usefulness of the idea and the tool and provided free feedback about the whole idea.

The subjects graded the feasibility in the range of 7 to 9, effectiveness in the range of 5 to 9 and usability in the range of 7 to 9. The grade 5 in effectiveness was given by one subject (Subject 5), where other feedback were in the range of 7 to 9. The grade 5 was given because Subject 5 thought that for a subject not much familiar with the system it would be harder to use it effectively. However, the subject with more experience and more familiarity with the system (Subject 1 and 3), thought they will be able to use it effectively and graded effectiveness as 9 and 8 respectively.

All the subjects think the tool is usable and the steps from selecting the fields to generating state machines were minimal and seemed easy enough and they would be able to use the tool without much help. A subject, however, suggested that feature to compare different state machines automatically and highlighting the changed behavior in one with respect to the other would be advantageous. This feature is indeed in my future work plan.

5.9.2 Round 2

Q2.1: *How difficult was it to select the functions / fields, on a scale of one to ten, for a given issue?*

Answer: Answering Q2.1, both subjects gave the procedure of selecting fields and functions a difficulty rating of two out of ten; with one being the easiest and ten the hardest. They stated that the keyword search is very effective in exploring the relevant fields and functions of an issue. The subjects stated that being the developer of the system this is the most basic information we can get from them. And even if a developer might not be familiar with all fields and functions, (s)he can guess their

names since (s)he is familiar with naming standards followed in the source code for defining fields and functions.

Q2.2: *After selecting the list of fields and functions, on a scale of one to ten, how difficult was it to come up with the valid constraints with respect to the scenario?*

Answer: Answering Q2.2, one of the developers graded the process of *selecting fields under different templates of constraints* a difficulty rating of one while the other graded two; with one being the easiest and ten hardest. The developers stated that since the fields are already selected by them, they have some idea of how to use them in a small number of templates.

Q2.3: *Do you think the newly added feature in the tool which displays a list of modified fields in a function automatically when you select that function (similarly, a list of functions modifying the field automatically load when you select the field) is advantageous over manually selecting both based on your knowledge?*

Answer: Both developers agreed that the feature is very useful and will greatly help developers once the system is put to use regularly.

Q2.4: *How many times, on an average, did you have to repeat the process (select the fields, functions select constraints generate state machine)to abstract program behavior according to your need and expectation?*

Answer: Both developers used the tool only once for each configuration, per issue; once for buggy version and once for the correct version as they were satisfied by the correctness of state machines and could see the symptoms of the issue in the first run.

Q2.5: *How much time did each go take on an average?*

Answer: The total time taken for the selection of fields and functions took less than a minute. Similarly, the process of defining constraints took less than a minute where one of the issues selected by the developers required as many as 5 constraints in one definition. However, the compilation and execution of the source code took most time which is about 5 minutes. The generation for both issues took less than a minute, and the subjects were able to spot the issue in under two minutes. Hence, the total time for one run of an issue took less than 10 minutes.

5.9.3 Round 3

In this round of interviews, as stated above, developers were asked to investigate two issues each. At the end of each investigation, the developers were asked for the suspected location of the issue and the important fields involved in the issue. All developers involved in this round gave correct answers for both sessions; manual debugging and with my tool. However, following questions were asked to get their feedback on the tool.

Observational Questions During the interview the following were observed while the developers were doing their tasks. The observations are provided directly in the summary table 5.9.

O 3.1.1: *How many times, on an average, did the developers have to repeat the process (select the fields, functions, select constraints and generate state machine) to abstract program behavior according to their expectation?*

Answer: All developers got the expected state machine, that showed their expected behavior, on the first time. Hence they did not repeat the process. However,

ID	**Subject 6** (Round 3)	**Subject 7** (Round 3)
Industry Experience (yrs)	2	5
Code Base Experience (/5)	0	2
O3.1.1	1	1
O3.1.2	Manual: 25 min Tool: 12 min	Manual: 12 min Tool: 10 min
O3.1.3	Code Comprehension, Forward Reasoning	Code Comprehension
Q3.2.1 (out of 10)	1	3
Q3.2.2 (out of 10)	1	3
Q3.2.3	Debug. Appr: Code Comprehension, Forward Reasoning Toolset: Visual Studio	Debug. Appr: Code Comprehension, Forward Reasoning, Backward Reasoning Toolset: Visual Studio
Q3.2.4	Beneficial in identifying location of bugs in complex scenarios	Beneficial as easily traces and points to the location of bug
Q3.2.5	No challenges	No challenges
Q3.2.6	F=10, E=9, U=8	F=9, E=8, U=9

Table 5.9: Summarized responses to interview questions in Round 3 (Part 1). F: Feasibility, E: Effectiveness, U:Usability

for demonstration purposes, as it took around 10 minutes to debug an issue with the tool, the team lead asked to try another issue and got his expected behavior on the first time too.

O 3.1.2: *How much time did the manual debugging take, and how much time was taken by each tool run, average if multiple?*

Answer: The time taken by developers to manually find the location of the issues ranged from 10 minutes to 30 minutes (mostly including code review and occasional

ID	**Subject 8** (Round 3)	**Mapped To (RQ)** (Round 3)
Industry Experience (yrs)	13	N/A
Code Base Experience (/5)	5	N/A
O3.1.1	1	N/A
O3.1.2	Manual: 8 min Tool: 11 min	N/A
O3.1.3	Backward Reasoning	N/A
Q3.2.1 (out of 10)	3	RQ2
Q3.2.2 (out of 10)	3	RQ2
Q3.2.3	Debug. Appr: Offline Analysis, Input Manipulation Toolset: Visual Studio, Valgrind	RQ3
Q3.2.4	Beneficial in analyzing system behavior and verifying effects of code changes	RQ3
Q3.2.5	Tool Maintenance	RQ4
Q3.2.6	F=N/A, E=8, U=8	RQ2, RQ3, RQ4

Table 5.10: Summarized responses to interview questions in Round 3 (Part 2). F: Feasibility, E: Effectiveness, U:Usability

partial executions). However, the time taken by the same developers for using the tool ranged from 10 minutes to 12 minutes. Note that this time also includes the full compilation and execution time of the test cases, which ranged from 4 minutes to 8 minutes per execution.

O 3.1.3: *Which diagnostic strategies were used?*

Answer: The three developers used a mix of three debugging strategies; Forward Reasoning, Backward Reasoning and Code Comprehension, as defined in [6]. In forward reasoning, the developer starts from the starting point of the execution and move towards the first occurrence of the issue in search of the cause of the issue, while

executing the source code. However, in backward reasoning, the developer starts from the first occurrence of the issue and moves backwards to the start of execution while searching for the cause of the issue. In code comprehension, developer reads through the code while making an understanding of the source code and try to find the issue with his mental picture of the code.

Q3.2.1: *How difficult was it to select the functions / fields in general on a scale of 1 to 10?*

Answer: Answering to the question, one of the developers graded the process of selection a difficulty level of one, while the remaining two graded it three out of ten.

Q3.2.2: *After selecting the list of fields and functions, on a scale of 1 to 10 how difficult was it to come up with the valid constraints with respect to the scenario?*

Answer: The grade (one) was selected by all three developers, for this question.

Q3.2.3: *What is your current practice/procedure of debugging the system or any other tool that is handy, when an issue is reported and assigned to you? Which tool support do you have?*

Answer: The answer to the question is divided into two parts; one in which I observed their practice during the manual debugging session and the other when they stated their normal debugging practice, dealing with everyday development issues. From my observation, one of the developers used Backward Reasoning by first reaching to the point where the issue can first be noticed and moving back to the location of the root cause, gradually. While the other two used Code Comprehension and Forward Reasoning to reach the location of the bug.

However, from their responses it was found that more experienced developers typically uses the debugging strategies of Input Manipulation and Offline Analysis, while the developers with entry to mid-level experience use a mix of Forward Reasoning, Backward Reasoning and Code Comprehension strategies while debugging an issue. In the input manipulation technique for debugging issues, the developer keeps modifying the input that is producing the wrong result and compare with the expected output, until he figures out the relation between input and output. In offline analysis, the developer relies on the post execution data such as log and execution traces to debug the issue.

Q3.2.4: *How beneficial is using this tool compared to your current practice?*

Answer: Since the current round of experiment involved developers with more experience than of the previous rounds, I got more concrete responses to this question than those in Round 1. The most senior developer said that the tool is useful for analyzing the behavior of the system and also for verifying the modification done to the system in subsequent releases. The tasks can be delegated to the developers who can verify their own changes by this tool. However, he thought that for debugging purposes, the tool is useful for entry to mid-level developers but may not be useful for highly experienced developers due to their familiarity with the code base over the tool (and state machines in general). One developer states that it seems useful as it can easily point the user to the actual location of the issue (function) to start their further investigation, and then come back to the tool for a more detailed state machine defined on the basis of the investigation results. The last developer stated that the tool looks very beneficial in situations where Backward Reasoning and Forward Reasoning is not

possible due to the unknown effects of the issue on the system or for more complex issues. He said that although as opposed to manual debugging it needs a full execution which takes more time, however he thought that in most situations the tool would take the developer to the location of the issue quicker than manual debugging practices.

Q3.2.5: *How easy is it to embed this tool into your current frameworks and infrastructure? Any challenges that you see in making this a part of your tool-set? Would you use it.*

Answer: In response to the question, the team leader stated that the biggest challenge in such tools is the maintenance, which is costly since the actual developers of such research tools move on. The other two developers stated that with any new tool there is a learning curve but they see no major issue adapting to this tool once included in their tool-set.

Q3.2.6: *Please provide your feedback about the feasibility, effectiveness, and usability of this tool and the idea of using your knowledge to break the issues down to fields and functions.*

Answer: In response to the question, the team leader stated that he wasn't sure about the feasibility of the tool as he has yet to explore it with other issues in the system. However, he graded the tool 8 for both effectiveness and usability. The other two developers graded the tool 10,9,8 and 9,8,9 for feasibility, effectiveness and usability respectively.

5.9.4 Summary of Interview Results: Answers to RQs

Table 5.7, 5.8 and 5.9 summarize all the 78 answers. The tables also map the questions to my original RQs.

In light of the feedbacks received from the subjects, I can conclude answers to my four RQs as explained above. For RQ1, I can safely conclude, after looking at responses to Q1.2.1 and Q1.3.2 in Table 5.7, that the state machines generated by my approach are correct and conform with normal program behavior. Responses to Q1.3.1, Q1.3.3, Q1.3.7, Q2.1, Q2.2, Q3.2.1, Q3.2.2 and Q3.2.6 conclude my RQ2. In light of the feedback, I can conclude that the main idea of selecting fields and functions is feasible, since all subjects graded the process of selection of fields and functions in a range of 1 to 3, and graded the feasibility of the approach in the range of seven to ten out of ten.

Responses to Q1.2.2, Q1.2.3, Q1.3.4, Q1.3.5, Q1.3.7, Q3.2.3, Q3.2.4 and Q3.2.6 answer my RQ3. From the responses, I found that combining the behavioral model with abstracted data gathered during real execution of program is useful for debugging and serves developer in more than program comprehension.

Finally RQ4 is answered by Q1.3.6, Q1.3.7, Q3.2.5 and Q3.2.6. Looking at the Table 5.7, most subjects didn't think they would face any trouble if they use the tool on regular basis. Those who found it difficult said the tool was worthy to learn, looking at the advantages it has to offer over traditional techniques. The subjects also think that the idea is effective and the tool is usable for them.

Moreover, as round 3 was specifically designed as a response to the reviews from the submission, I found that since the developers were also able to find the relevant

fields and location of the issues with just looking at the buggy state machines, the results of first round is also valid.

5.10 Threats to Validity

There are several typical threats to the validity of this study. In this section, I will explain the most important threats and my approach to tackle them.

The tool developed as an outcome of this research is developed, tested and configured by one researcher. There might be some bugs in the tool that are yet to be found. However, the output of the tool, i.e. the state machines, were constantly validated by our contact lead developer at the company. Hence, I can safely assume that the state machines generated by the tool are correct, as pointed out by our contact lead developer as well as the interviewees.

Another threat to the validity is being too specific to my program under analysis (Autopilot) and/or the company's debugging process. However, Autopilot is similar to any other industry standard safety critical embedded systems. Hence, the idea can directly be applied to any other system maintaining global state fields with very little modification in the supporting tool. In addition, the company's debugging process is very typical and not specific to this company at all.

The bugs used in this study are real bugs from the history of the software that are selected with the help of our industry partner to be representative of their typical bugs. However, they can also be reasonably generalized to a broader context, since my approach is not designed for a specific domain's bugs. The state machines however, work on the function call level. Thus, the smallest unit that I can localize the bug is

the last function in a call stack that contains the bug.

Another validity threat is to do with generalization of the interview results. Ideally, more subjects should have been interviewed. Given that the interviews are expensive for the company, there is a plan to have a more extensive study with students.

Chapter 6

Conclusion

Fully automated specification mining techniques have been studied in literature for many years but they have not been used in industry much yet. One of the main reasons is that it is hard for the automated abstraction techniques to generate models in the exact level of details that the user needs. Moreover, while passive inference techniques use program execution traces to infer its behavior, a diverse set of test cases or scenarios are required to capture a complete state machine. Active inference techniques come to help here by tackling the limitation by asking queries, but since they require extensive involvement of domain experts, the maintenance of such is a concern for companies who have limited resources.

In this thesis, I proposed an interactive approach for specification mining, where the domain knowledge of the users and their required level of details are collected as inputs in the form of fields and functions which is a basic knowledge developers are supposed to have while working on any code base. The tool is focused on debugging application of specification mining and let the users to easily change the perspective

and or level of details by selecting different fields to monitor and constraints to use. Our approach has been verified in a full size industry setting and its evaluation in terms of a series of interviews confirmed its feasibility and usefulness, compared to the current state of practice, in debugging. Although developed in collaboration with Micropilot, the tool is highly configurable to be used with any code base developed in c/c++ and maintained as Visual Studio project.

6.1 Limitations

Since the approach is fully dependent on state fields, one of the limitations of the approach is that it can only be applied to embedded systems that either maintain their fields globally, or use struct instances to maintain state fields and pass its reference around to maintain state fields.

Another partial limitation of the tool is that it needs the subject code base to be maintained as a Visual Studio project as its profiler depends on debugging switches that are configurable with Visual Studio. However, the limitation can be overcome by segregating the profiler from the tool and traces can be fed manually to the abstractor rather than through an automated process.

6.2 Future Work

We are planning to take at elast two directions in our future work as follows:

6.2.1 Extended User Study

One of the future directions for the project is extending the user study and evaluations, in terms of number of participants and more issues per user. As this is not feasible with company employees due to cost associated with the interviews, a secondary, more extensive, user study is scheduled in near future, where we interview students on a dummy project (a Tetris game is selected).

Since the tool has been developed as a configurable system which can be hooked up with any Visual Studio solution, we can replicate it using open source subject projects with student participants.

We plan to have a more in-depth evaluation of the tool, where the participant debug more issues with different levels of difficulties.

6.2.2 Fault Augmentation

Augmenting the state machine with more bug related information is very useful for the developers and testers to look at the specification model of a system and easily understand what parts or which kind of scenarios are more error-prone. This high-level idea of augmenting state machines with fault information is similar to defect prediction studies, where one links a bug tracking system with the corresponding version control system to identify files or modules that had the most defects in the past and past faults are analyzed automatically. An overview of the methodology can be viewed in Figure 6.1. In a nutshell, the process is divided in to following steps.

Extract Failing Test Cases

In this step, I look in the bug tracking history to find the faults previously en-

countered in the system. I gather the last `n` test cases with true failures.

Finding corresponding versions

For this step, I link the source code versions to the faults based on their timestamps in the version control and defect tracking systems.

Executing the test cases on their corresponding versions

In this step, I perform step 1 of my main proposed methodology to extract the execution traces of the failing test cases, by running them on their corresponding versions that they originally failed on.

Abstracting and merging failing behavior into failing state machines

In this step, I abstract and merge the failing test cases traces, similar to what was explained in the previous section. This is done individually per test case, so that we have a state machine (lets call them failing state machines) for each failing test cases.

Maintaining a counter for failing states and transitions

Finally, I compare the states and transitions in the failing state machines and the correct state machine (generated by abstracting the current set of passing tests' execution traces). There is a historical failure counter (HFC) assigned to each model entity (state and transition) in the original state machine. HFC of an entity is equal to the number of times the same entity has been in the failing state machines. For instance, if we see the same state A of the original state machine in 5 failing state machines, HFC of state A is 5. This counter will be used as a representation of how likely this state or transition was to contribute to a failing scenario. According to [11] and other defect prediction studies, this is also a good predictor of future failures.

This direction has already been started and we have some progress. The fault

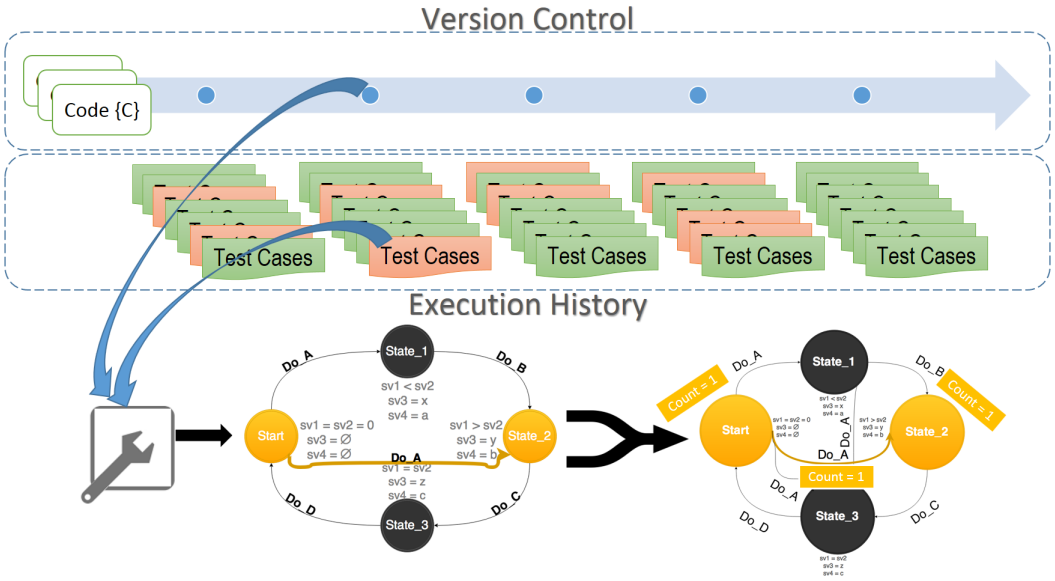


Figure 6.1: Overview of Fault Augmentation

augmentation is already implemented. However, the initial findings on applying the feature with the real issues reported many false positives, since most failing and passing test cases share many common behaviours. So more work on this direction is needed.

6.2.3 Potential Improvements

Defining more templates for the abstractor The tool can be improved to capture more detailed requirements by defining more templates for constraints generation. Currently users are asked to define constraints related to one, two or three state fields, while the tool limiting the observed relationship to "less than", "equal to" and "greater than". More templates can be developed where users enter more complex constraints and apply mathematical and statistical operations, including ad-

dition of more relationships and complex equations of fields involving n number of fields in one template.

Regression Anomaly Detection Moreover, a potential future direction for the research project is automatic comparison of behavioral models that could help even more in debugging processes. For instance, for regression testing, the tool could be automated to generate and compare the state machines, automatically, whenever a new check-in event is triggered (in the version control system). This would help the users in verifying the effect of their modification after each check-in, as a precaution in case the changes from one user conflict with the changes from another, and changing the expected behavior of the system, and would also help in timely updating of behavioral models of the system.

Appendix A

Profiler

A.1 AspectC++

AspectC++ [2] is the language extension of C++ to incorporate AOP with C and C++ code bases. Despite the limited online support available for the tool at the time, it was incorporated successfully with a few open source projects that were obtained from GIT. AspectC++ would also work with small modules of Micropilot code base. Listing A.1 provides an example of a dummy source file including aspects. However, on trying aspectC++ with the Micropilot's huge code base of the Autopilot project, the compiler couldn't compile the weaved code, even with the help of makefiles that were already created by the company. The reason found for this after investigation was that after weaving aspects in the code base, its compiler (ag++) compiles both C and C++ code files with g++ command. In practice, g++ can compile both c and c++ programs. However, possibly due to implicit pointer casts or C-99 style initialization of structs, the actual C files in the Micropilot's code base could only

be compiled with gcc command. This was also validated by our contact person at Micropilot.

```
#ifndef __trace_ah__
#define __trace_ah__
#include <cstdio>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
using namespace std;

aspect trace {
  int depth=-1;
  advice execution ("% ...::%(...)" && !"% ...::%(void)") : before ()
      {
    stringstream ssSt;
    stringstream ssDfn;
    std::ofstream outfile;
    outfile.open("out.txt", std::ios_base::app);
      //outfile << "Data";
        depth++;
    /**************INDENTATION*****************/
    //for(int i=0;i<depth;i++)
    //  outfile << "\t";
    /**************INDENTATION*****************/
          outfile << "Entering;" << JoinPoint::signature() << ";(";
    for (unsigned i = 0; i < JoinPoint::args(); i++) //printvalue(tjp->
   arg(i), JoinPoint::argtype(i)); //THIS WAS IN THEIR GUIDE BUT NOT
   WORKING
    {
      if(strcmp(JoinPoint::argtype(i),"i") == 0 || strcmp(JoinPoint::
   argtype(i),"l") == 0)
      {
        outfile << *(int *)tjp->arg(i);
        //printf("%d", *(int *)tjp->arg(i));
        if(i+1 < JoinPoint::args())
          outfile << ",";
      }
      else if(strcmp(JoinPoint::argtype(i),"c") == 0)
        {
        outfile << *(char *)tjp->arg(i);
        //printf("%c", *(char *)tjp->arg(i));
        if(i+1 < JoinPoint::args())
          outfile << ",";
        }

      else
      {
        outfile << "datatype=" << JoinPoint::argtype(i);
        if(i+1 < JoinPoint::args())
          outfile << ",";
      }
    }
    outfile << ");";
    /*************GLOBAL VARIABLES*****************/
    outfile << variableInTheSubjectCodeBase;
    /*************GLOBAL VARIABLES*****************/
```

```
54      outfile << JoinPoint::filename() << endl;
55        }
56    advice execution ("% ...::%(void)") : before ()
57        {
58      std::ofstream outfile;
59      outfile.open("out.txt", std::ios_base::app);
60          depth++;
61      /**************INDENTATION*****************/
62      //for(int i=0;i<depth;i++)
63      //  outfile << "\t";
64      /**************INDENTATION*****************/
65            outfile << "Entering;" << JoinPoint::signature() << ";;;;";
66      /*************GLOBAL VARIABLES*****************/
67      outfile << variableInTheSubjectCodeBase;
68      /*************GLOBAL VARIABLES*****************/
69      outfile << JoinPoint::filename() << endl;
70        }
71        advice execution("% ...::%(...)" && !"void ...::%(...)") : after()
72        {
73      stringstream ssSt;
74      stringstream ssDfn;
75      std::ofstream outfile;
76      outfile.open("out.txt", std::ios_base::app);
77      JoinPoint::Result res = *tjp->result();
78      /**************INDENTATION*****************/
79      //for(int i=0;i<depth;i++)
80      //  outfile << "\t";
81      /**************INDENTATION*****************/
82      outfile << "Exiting;" << tjp->signature()<< ";"<< res << ";;;";
83      /*************GLOBAL VARIABLES*****************/
84      outfile << variableInTheSubjectCodeBase;
85      /*************GLOBAL VARIABLES*****************/
86      outfile << JoinPoint::filename() << endl;
87      depth--;
88        }
89    advice execution("void ...::%(...)") : after()
90    {
91      stringstream ssSt;
92      stringstream ssDfn;
93      std::ofstream outfile;
94      outfile.open("out.txt", std::ios_base::app);
95      /**************INDENTATION*****************/
96      //for(int i=0;i<depth;i++)
97      //  outfile << "\t";
98      /**************INDENTATION*****************/
99      outfile << "Exiting;" << tjp->signature() << ";;;;";
100     /*************GLOBAL VARIABLES*****************/
101     outfile << variableInTheSubjectCodeBase;
102     /*************GLOBAL VARIABLES*****************/
103     outfile << JoinPoint::filename() << endl;
104     depth--;
105   }
106 };
107   #endif
```

Listing A.1: sample aspect source code for execution tracing

A.2 Entry and Exit Hook Functions

As mentioned in the section 4.1 the references of entered and exited functions and their arguments are extracted from the registers by manipulating the program context in the registers. The functions are defined in a c++ library project, which can be linked with any c/c++ project to obtain execution traces.

Listing A.2 shows the corresponding source code. The manipulation starts by pushing all the registers to the stack. Then the stack pointer is moved by 32 bytes to address the newly added data. Finally, the function that receives the stack pointer in its first argument is called. The top of the stack contains the address to the current line of code (in the subject code's function that is just entered). Since the subject function is just entered, and the instruction calling the function is 5 bytes long, this amount can be subtracted from the value at the top of stack to get the address of the current function call.

The values of the function parameters can also be accessed in a similar way. For parameter `i` of the function, if the parameter is a numeric or character value, the value is stored at location [2 + `i`] of the stack. If the parameter is a pointer (for instance, string or instance of a struct), its reference is stored in the given location which can be used to print the value of the parameter.

```
1  extern "C" __declspec(naked) void __cdecl _penter()
2  {
3    _asm
4    {
5      pushad                // save all general purpose registers
6      mov    eax, esp       // current stack pointer
7      add    eax, 32        // calculate the pointer to the return address
       by adding 4*8 bytes. (8 register values are pushed onto stack which
       must be skipped)
8      push   eax            // push pointer to return address as parameter
       to EnterFunc
9
10     call   EnterFunc
11
```

```
12      popad                   // restore general purpose registers
13      ret                     // start executing original function
14    }
15  }
16
17
18  extern "C" __declspec(naked) void __cdecl _pexit()
19  {
20    _asm
21    {
22      pushad                  // save all general purpose registers
23      mov     eax, esp        // current stack pointer
24      add     eax, 32         // calculate the pointer to the return address
      by adding 4*8 bytes. (8 register values are pushed onto stack which
      must be skipped)
25      push    eax             // push pointer to return address as parameter
      to ExitFunc
26
27      call    ExitFunc
28
29      popad                   // restore general purpose registers
30      ret                     // start executing original function
31    }
32  }
```

Listing A.2: *_penter* **and** *_pexit* **function definitions**

In addition to the references to the selected function, references to three more functions are passed in **step 3** of Figure 4.1. The purpose of these references are to, (1)-indicate where tracing should start, (2)-obtain the reference to the state struct instance and (3)-indicate the end of execution. The reason for this is explained below. The print statements for state fields are automatically generated by the static analysis tool explained in section 4.1 and appended to the profiler. The profiler is then hooked to the solution through an automated script and both projects are compiled.

Referring to the Listing A.3 and the function references explained in the above text, when the subject program is executed, the reference to the function (1) at the entry point indicates the starting point of the execution and the tracing starts here. For every following function entry, the EnterFunc function inside the profiler is called with the current stack pointer. The address of the function is searched inside the list passed to the profiler in **step 3**. If it exists, the function is traced. Similarly, on

entering the function whose reference is contained in the second additional reference (2), the reference to the global instance of struct is saved. In fact, the referenced function is called with the struct instance as one of its arguments. The saved reference to the instance is used by the PrintStateFields function to print the values. The library maintains a buffer of 100KB of trace lines before writing to the file which reduced the system interrupts and optimizes the performance of the profiler. The buffer size can be increased in configuration to further optimize the profiler in the cases when a large number of functions are selected. If the execution is finished and the buffer still has some data, the reference to the last of the additional functions helps the profiler in dumping the remaining lines of traces to the trace file.

On the invocation of ExitFunc, the address in the stack pointer refers to the last line of the function exiting in the subject code base, where its information is not readily available. However, to trace the function name we need the address to the first line of function which is also passed from the main file. One possible solution for this is getting the size of each function on the run-time and subtracting it from the address of last line. The option works for most cases since the static analyzer can take care of the order of the functions they are declared at. However, it doesn't work if there are declarations of structs or enumerations between function declarations.

The alternative approach is using the **DebugHlp** library which is shipped with all versions of Windows. Using the `SymFromAddr` function from the mentioned library, one can get the name of the symbol from its address, for a function or a struct instance. However, since this is a debug library, and although the library can be used with release mode binaries, function calls to this library are expensive and

can affect the performance of the original execution. Hence, calls to this function need to be optimized by only calling it once for any function called in the subject program, including the primitive functions such as printf, scanf etc. Please refer to the ExitFunc method in Listing A.3 to view the optimized code. Another limitation of the DebugHlp library is that it is not thread safe. Hence if an application is multithreaded, the above profiler will only generate complete execution traces of the first thread while the remaining traces will only contain entry method lines.

```
1
2  bool startTracing = false;
3  HANDLE         process;
4  state_struct * address = 0;
5  char traceFile[200] = "C:\\Automated-Traces\\Configuration-Date-
       ExecutionScenario.log";
6
7  void _stdcall ExitFunc0(unsigned * stack)
8  {
9      if(startTracing){
10     if (IsExistInIgnoringReferenceList(stack[0]))
11       return;
12     SYMBOL_INFO  * mysymbol;
13     Signature * f = FuncTable;
14     bool found = false;
15     while (f->function)
16     {
17       if (f->endAddress != NULL && (void *)stack[0] == f->endAddress)
18       {
19         found = true;
20         linesBuffer = linesBuffer + "Exiting " + f->name + ";\n";
21         if (address != 0)
22           PrintStateFields(0);
23         break;
24       }
25       f++;
26     }
27
28     if (found)
29         return;
30     mysymbol = (SYMBOL_INFO *)calloc(sizeof(SYMBOL_INFO) + 256 * sizeof(
       char), 1);
31     mysymbol->MaxNameLen = 255; mysymbol->SizeOfStruct = sizeof(
       SYMBOL_INFO);
32     SymFromAddr(process, (DWORD64)((void *)stack[0]), 0, mysymbol);
33     char temp[MAX_TEMP_LENGTH];
34     strcpy_s(temp, "                                                    ");
35     wsprintf(temp, "%s", mysymbol->Name);
36     if (my_strstr(mysymbol->Name, "std:") || my_strstr(mysymbol->Name, "
       el:") || my_strstr(mysymbol->Name, "printf") || my_strstr(mysymbol->
       Name, "scanf")){
37       insertReferenceToIgnore(stack[0]);
38       return;
```

```
39      }
40      f = FuncTable;
41      if (mysymbol->Name[0] != '\0')
42        while (f->function)
43        {
44          if (mystrcmp(f->name, temp))
45          {
46            found = true;
47            f->endAddress = (void *)stack[0];
48            linesBuffer = linesBuffer + "Exiting " + f->name + ";\n";
49            if (address != 0)
50              PrintStateFields(0);
51            break;
52          }
53          f++;
54        }
55      if (!found)
56      {
57        insertReferenceToIgnore(stack[0]);
58      }
59      free(mysymbol);
60    }
61  }
62
63  void _stdcall EnterFunc0(unsigned * stack)
64  {
65      if (ProcessDetachReference == (void *)(stack[0] - 5))
66    {
67      WriteToFile(true);
68    }
69
70    if (isFirstLoad && initFunctionReference == (void *)(stack[0] - 5))
71    {
72      process = GetCurrentProcess();
73      SymInitialize(process, NULL, TRUE);
74      isFirstLoad = false;
75      FILE * pFile = fopen(traceFile, "w+");
76      time_t now = time(0);
77      tm* localtm = localtime(&now);
78      fwrite("Start Time: ", sizeof(char), 12, pFile);
79      fwrite(asctime(localtm), sizeof(char), strlen(asctime(localtm)),
       pFile);
80      fwrite("\n", sizeof(char), 1, pFile);
81      fclose(pFile);
82      startTracing = true;
83      return;
84    }
85    if (once && stateFieldsInitReference == (void *)(stack[0] - 5))
86    {
87      address = (ac_state *)stack[2];
88      once = false;
89    }
90    if (startTracing) {
91      void      * pCaller = (void *)(stack[0] - 5); // the instruction for
       calling _penter is 5 bytes long
92      Signature * funct = FuncTable;
93      while (funct->function != NULL)
94      {
95        if ((void *)(stack[0] - 5) == funct->function)
96        {
97          if (address != 0)
```

```
98              PrintStateFields();
99            linesBuffer = linesBuffer + "Entering " + funct->name + ";\n";
      break;
100
101          }
102          funct++;
103        }
104     }
105  }
106
107  void PrintStateFields()
108  {
109    linesBuffer = linesBuffer + "field1=" + std::to_string(((state_struct
       *)address)->field1) + ";";
110    linesBuffer = linesBuffer + "field2=" + std::to_string(((state_struct
       *)address)->field2) + ";";
111    linesBuffer = linesBuffer + "\n";
112    WriteToFile(false);
113  }
```

Listing A.3: EnterFunc and ExitFunc function definitions

A.3 Trace Processor

Listing A.4 demonstrates the scenarios encountered when processing a trace file in **step 6**. The first scenario can be seen in the trace from line 1 to 8. While processing the trace file, Function1 is pushed to stack on reading Entering Function1, with the state values in the line before. Then Function2 is pushed with the values on Entering Function2. On reading Exiting Function2, Function2 is popped from the stack after comparing the field. Since the values of state fields in the lines after Exiting Function2 and before Entering Function2 are the same, Function2 is not moved to the processed trace file. While reading Exiting Function1 at line 7, Stack is popped and "before enter Function1" and "after exit Function1" values are compared. Since the values are different, Function1 is recorded in the processed trace file with the the modified values. Similarly, for second scenario from line 9 to 16, when popping Function2 the values at line 11 and 14 are compared. Since they are different, Function2 is

recorded in the processed trace file. The third scenario can be seen from line 17 to 24. Processing this scenario, Function1 will be recorded in the processed trace file as the one responsible for the changes made to the state fields.

```
1  field1=0, field2= 0
2  Entering Function1;
3      field1=1, field2= 1
4      Entering Function2;
5      Exiting Function2;
6      field1=1, field2= 1
7  Exiting Function1;
8  field1=1, field2= 1
9  field1=1, field2= 1
10 Entering Function1;
11     field1=1, field2= 1
12     Entering Function2;
13     Exiting Function2;
14     field1=2, field2= 2
15 Exiting Function1;
16 field1=2, field2= 2
17 field1=2, field2= 2
18 Entering Function1;
19     field1=2, field2= 2
20     Entering Function2;
21     Exiting Function2;
22     field1=2, field2= 2
23 Exiting Function1;
24 field1=3, field2= 3
```

Listing A.4: Sample Trace

Bibliography

[1] GH. https://msdn.microsoft.com/en-us/library/xc11y76y.aspx. Last accessed 2016-05-16.

[2] AspectC++. https://www.aspectc.org/. Last accessed 2016-05-07.

[3] clang. https://clang.llvm.org/. Last accessed 2017-05-16.

[4] Z. A. Al-Sharif. *An Extensible Debugging Architecture Based on a Hybrid Debugging Framework*. PhD thesis, University of Idaho, 2009.

[5] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-state Machines from Samples of Their Behavior. *IEEE transactions on Computers*, 100(6):592–597, 1972.

[6] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. Where is the Bug and How is it Fixed? An Experiment with Practitioners. In *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2017, pages 1–11, 2017.

[7] L. C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of

Uml Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9):642–663, Sep 2006.

[8] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. Modisco: A Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8): 1012–1032, Aug 2014.

[9] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension Through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, Sep 2009.

[10] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24. ACM, May 2006.

[11] M. D'Ambros, M. Lanza, and R. Robbes. An Extensive Comparison of Bug Prediction Approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, May 2010.

[12] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko. Fsm-based Conformance Testing Methods: A Survey Annotated with Experimental Evaluation. *Information and Software Technology*, 52(12):1286–1297, May 2010.

[13] 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. *Science of Computer Programming*, 69(1):35–45, Dec 2007.

[14] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based Diagnosis of Hardware Designs. *Artificial Intelligence*, 111(1):3–39, Jul 1999.

[15] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349. ACM, Nov 2008.

[16] M. Gabel and Z. Su. Symbolic Mining of Temporal Specifications. In *Proceedings of the 30th international conference on Software engineering*, pages 51–60. ACM, May 2008.

[17] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight Cross-Project Anomaly Detection. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 119–130. ACM, Jul 2010.

[18] S. Hassan, U. Qamar, T. Hassan, and M. Waqas. Software Reverse Engineering to Requirement Engineering for Evolution of Legacy System. In *IT Convergence and Security (ICITCS), 2015 5th International Conference on*, pages 1–4. IEEE, Aug 2015.

[19] D. Hiebert. Exuberant ctags, 1999.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *ECOOP'97Object-oriented programming*, pages 220–242, 1997.

[21] M. Kim and A. Petersen. An Evaluation of Daikon: A Dynamic Invariant Detector, 2004.

[22] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the sbbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*, pages 1–12. Springer, 1998.

[23] D. Lo and S. Maoz. Scenario-based and Value-based Specification Mining: Better Together. *Automated Software Engineering*, 19(4):423–458, Dec 2012.

[24] D. Lo, S.-C. Khoo, J. Han, and C. Liu. log4Net. `https://logging.apache.org/log4net/`. Last accessed 2017-06-15.

[25] D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining Software Specifications: Methodologies and Applications*. CRC Press, May 2011.

[26] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, May 2008.

[27] H. Malik, H. Hemmati, and A. E. Hassan. Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1012–1021. IEEE Press, May 2013.

[28] S. Maoz. Using Model-based Traces as Runtime Models. *Computer*, 42(10), Oct 2009.

[29] C. Mateis, M. Stumptner, and F. Wotawa. Modeling Java Programs for Diagnosis. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 171–175. IOS Press, Aug 2000.

[30] W. Mayer and M. Stumptner. Model-based Debugging–state of the Art and Future Challenges. *Electronic Notes in Theoretical Computer Science*, 174(4): 61–82, May 2007.

[31] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static Specification Inference Using Predicate Mining. In *ACM SIGPLAN Notices*, volume 42, pages 123–134. ACM, Jun 2007.

[32] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting Developers of Big Data Analytics Applications When Deploying on Hadoop Clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 402–411. IEEE Press, May 2013.

[33] D. Spinellis. Effective debugging. 2017.

[34] R. Taylor, M. Hall, K. Bogdanov, and J. Derrick. Using Behaviour Inference to Optimise Regression Test Sets. In *IFIP International Conference on Testing Software and Systems*, pages 184–199. Springer, 2012.

[35] A. Valdes and K. Skinner. Adaptive, Model-based Monitoring for Cyber Attack Detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 80–93. Springer, Aug 2000.

[36] T. Vogel, A. Seibel, and H. Giese. Toward Megamodels at Runtime. In *Proceedings of the 5th International Workshop on Models@ run. time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway*, volume 641, pages 13–24, 2010.

[37] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, 2006.

[38] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse Engineering State Machines by Interactive Grammar Inference. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 209–218. IEEE, October 2007.

[39] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[40] H. Zhong, L. Zhang, and H. Mei. Inferring Specifications of Object Oriented Apis from Api Source Code. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*, pages 221–228. IEEE, Dec 2008.

[41] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. Le Traon. Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1064–1071. ACM, Mar 2014.