A SELF-ADAPTIVE FRAMEWORK FOR FAILURE AVOIDANCE IN CONFIGURABLE SOFTWARE

by

Jacob Swanson

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Myra B. Cohen

Lincoln, Nebraska

May, 2014

A SELF-ADAPTIVE FRAMEWORK FOR FAILURE AVOIDANCE IN CONFIGURABLE SOFTWARE

Jacob Swanson, M.S.

University of Nebraska, 2014

Adviser: Myra B. Cohen

Many software systems today are highly-configurable, meaning the user can customize their instance of the application, adding and removing features as needed. However, this configurability makes it harder to ensure that faults will not escape testing and manifest as failures in the field. This is because many failures are configuration-dependent; they will appear under certain combinations of features configurations, but not others. Once faults are found and reported in the field, there can be a long lag time (days, weeks or even months) until patches are created and released to fix them. In the meantime, users will continue to use these software systems and need to be able to workaround potential failures. This thesis presents a self-adaptive framework for avoiding configuration-dependent failures at runtime. The framework is built on top of an existing self-adaptive system, Rainbow, and is implemented as a distributed client-base, with a central controller for applying the failure avoidance reconfigurations to clients as needed. Over time as the system gains global information of failures that have occurred, it will not only adapt to new configurations, but it guards other clients from these potentially bad configurations as well.

We have performed two evaluations to determine the feasibility of this framework in practice. We first evaluate a set of heuristic failure avoidance algorithms which forms the core of our adaptation, and compare these against an existing bounded brute force algorithm. We find that our heuristic algorithms are effective in avoiding most

failures, but for a lower computational cost. Although we miss some workarounds found by the brute force algorithm, we also find new workarounds that are outside of the bounded search space suggesting that a heuristic approach may be better. We then implemented our framework on top of Rainbow and evaluate this on the Firefox web browser. In a study using four distributed instances of Firefox and a set of seeded faults based on real Firefox configuration bugs, we determine that we can workaround them all, and that once observed, we will no longer see the same failure again. We also see new native failures that we did not seed, and find workarounds for these as well. We conclude that our self-adaptive framework is feasible approach for avoiding configuration dependent failures at runtime.

ACKNOWLEDGMENTS

First off I want to thank my advisor, Professor Myra Cohen for her support and dedication during my research. Without her help my work would never have made it this far. I also want to thank Professor Matthew Dywer for his insight and advice on this research. His questions and observations have helped immensely with shaping my work for the better. A large thank you as well to Professor Matthew Dwyer and Professor Sebastian Elbaum for agreeing to serve on my committee and offering your feedback and advice.

I want to specifically thank Brady Garvin for putting up with my questions about code he wrote years ago. Without his assistance providing artifacts and modifying his algorithm to work in this framework this thesis would not have been completed. I also want to thank all of the professors who have contributed to my education over the course of my college career.

To all of the members of the EsquaRED lab, thank you. The time I have spent in your presence has been quite wonderful, and I can't imagine trading it for anything.

Finally I want to say thank you Jenn. Without you I would never have sent an email to Professor Cohen, never have taken an internship that led to graduate school, and never have written this thesis.

GRANT INFORMATION

This work was supported by the AFOSR through award FA9550-10-1-0406 and by the NSF through awards CCF-1161767 and CCF-0747009.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many programs offer the user the option of selecting a custom set of features (or configuration options) when running an application. For instance, a typical web browser provides options to open web pages in new tabs or within the same window, and compilers may include different kinds of optimization flags. These features may improve the robustness of a system, improve performance, or simply allow for increased customization. The existence of large numbers of features means that software testing, which is an already expensive process, will be harder and may miss many of the less commonly used customizations [44]. A typical configurable system such as a web browser may have hundreds or thousands of options which leads to billions or trillions of possible customizations [14]. Research on testing configurable systems has focused on a particular type of fault called a *feature interaction fault*; a fault that occurs only under a specific combination of features [14, 25, 34, 44, 48, 55]. The problem of feature interaction faults has existed for years, with early work arising from the telecommunications industry [57]. These types of faults are difficult to detect and understand because of interlocking dependencies. [25, 44]. Patching a feature interaction fault can involve digging deep into the source code to look for problems that are not obvi-

ous. And reproducing it can mean having to select the correct configuration, which is non-trivial when there are billions of choices [14, 55]. Because feature interaction faults are hard to detect they can often slip into released software. And once there, it may take a long time to detect, and then once detected to repair [3].

Recent work on self-adaptive software [3, 6, 10, 16, 20, 24, 29, 49] may provide a way to allow configurable systems to continue running, despite the existence of feature interaction faults. In self-adaptive software systems, the system is run within a framework that monitors, analyzes, plans and then executes change. This is called the MAPE loop [6, 53]. A *utility function* measures specific quality attributes of the running system such as response time, system load or network bandwidth. These are then monitored and when a change is detected that exceeds some threshold, the system analyzes this change, creates a plan and then executes a reconfiguration. Traditionally, MAPE has been applied only to continuous utility functions of the system – primarily those for quality attributes. But recent work has suggested that we can also create self-adaptation in response to discrete changes such as system failures [4, 28]. In [4] Carzaniga et al. present a technique to automatically re-write small parts of a program (i.e. an automated patch), providing *workarounds* for failures seen in web applications. Furthermore, Garvin et al. [25, 28] present a set of algorithms, and performed a feasibility study to show that it is possible to avoid failures in configurable systems when they are only manifested under a specific set of feature combinations (i.e. they are *configuration dependent*). They also showed that there was *locality* of the combinations of features meaning that when we avoid one failure, we are likely to avoid additional ones.

In this thesis we build upon the research of Garvin et al. to implement a self-adaptive framework for avoiding configuration dependent failures. First, we experiment with some variations of their failure avoidance algorithms and evaluate these

alternatives with respect to effectiveness and efficiency. We then propose a framework that modifies an existing self-adaptive framework, Rainbow [23], for our purpose. We extend Rainbow with new methods to monitor for discrete failures, and add our strategies for avoiding and guarding against these failures. We then apply this framework to a distributed Firefox installation. We evaluate its effectiveness on a set of real faults for the Firefox web browser. We show that self-adaptation for avoidance of configuration dependent failures is feasible and that over time our system will be able to both avoid and guard against future failures.

1.1 Motivation

Consider the situation in which a user is browsing the Internet. As all users can attest, failures occur and can sometimes seem random. The screen hangs up, the browser unexpectedly crashes, these are common parts of a users' experience. The simplest response is to restart, and continue as normal, hoping the problem is not encountered again. A proactive user might be inclined to submit a bug report to the software owners, but responses can be slow and patches may be shelved until the next update [3].

One of the reasons for these types of problems to escape testing is that browsers are highly configurable systems and its impossible to comprehensively test all the different combinations of configurations with all of the different use cases [14, 25, 34, 44, 48, 55]. This can lead to a situation where an error exists because of some combination of configurations, and changing the configurations avoids the error. This set of reconfigurations effectively becomes a workaround for that particular error. A good workaround is one that does not overly impact the functionality of the intended use case. This means that workaround should only include the reconfigurations necessary to avoid

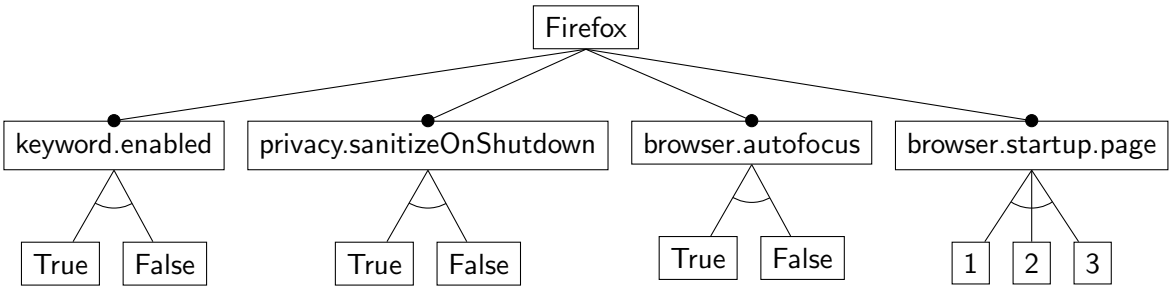

Figure 1.1: Firefox Feature Model

the failure. By keeping the number of reconfigurations (the distance) between the workaround and the starting configuration small the impact on the functionality will be minimized.

Instead of waiting for a developer to fix the bug, a user could search for a workaround using alternative configurations. This of course requires that the bug can be replicated, and the user has a large amount of free time available. But what if the work of searching for a workaround could be done behind the scene? Now when an error is encountered, the current configuration and the steps necessary to replicate the error are sent to a failure avoidance algorithm. If the failure can be replicated, then the failure avoidance algorithm can search for a set of reconfigurations that avoid the failure. If a passing configuration is found, then this becomes a workaround and is sent back to the user, and their configuration is updated with reconfigurations in the workaround. The failure avoidance algorithm can update the user's guard which will prevent the failure from occurring in the future by monitoring the configuration and preventing reconfigurations that would return the user to the original failing configuration state Finally, this workaround can be deployed to all users of a system, protecting others from ever experiencing the same failure.

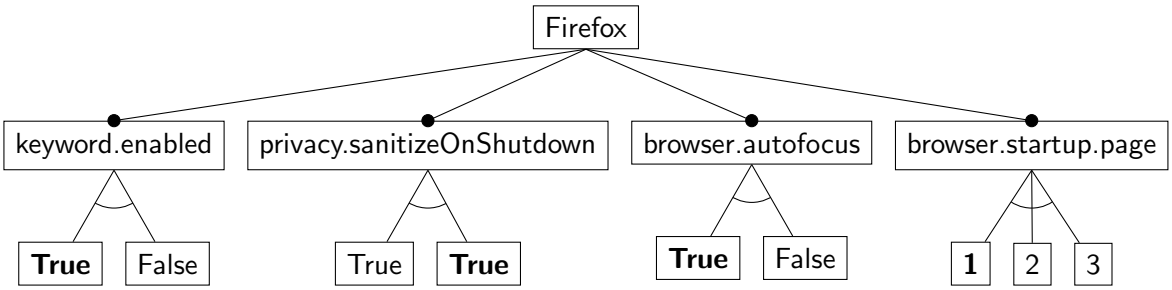

Figure 1.2: Firefox Default Configuration

1.1.1 Example

Consider a small part of the feature model for Firefox as is shown in Figure 1.1. In this figure we see four configuration options. The first three *keyword.enabled*, *privacy.sanitizeOnShutdown*, and *browser.autofocus* are all optional, binary features. We can include them (`True`) or exclude them. The last feature, *browser.startup.page* is mandatory and has three possible values (1, 2 or 3). The system will start in a default configuration. But there are 24 possible configurations that this system can reconfigure to, and the user is allowed to change these at will. Lets look at the case where a user is in the default configuration (see Figure 1.2). That means that *keyword enabled* is set to `True`, *privacy.sanitizeOnShutdown* is set to `False`, *browser.autofocus* is set to `True`, and *browser.startup.page* is set to 1.

As time goes on the user has made some changes to the system, and now *browser.autofocus* is set to `False` and *browser.startup.page* is set to 3. This configuration is shown in Figure 1.3.

As the user continues browsing, everything is running normally, but then the user runs into a problem. A failure occurs under their specific use-case (or test case). The first obstacle to overcome is recreating the failure. Using a web browser as an example we would need to capture all of the actions a user has made leading up to the failure. This will be represented as a test case. If, by running the test case with the current

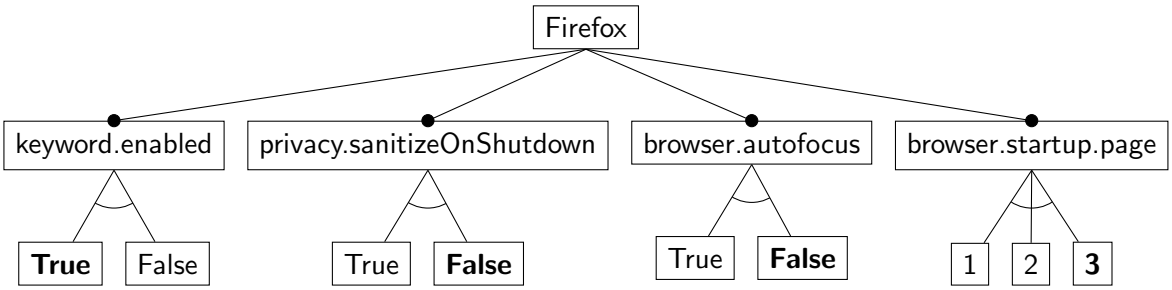

Figure 1.3: Firefox Feature Model Selected Configuration

configuration the failure is reproduced then we can search for a new configuration where this failure is no longer seen. If for any reason the failure is not replicated, then the failure avoidance algorithm will be unable to search for a workaround. In this particular example, we find that the problem is caused by the change the user made to *browser.startup.page*, setting it to 3.

We have two primary goals when looking for a workaround. First, we want to stay as close as possible to the user's starting configuration so that they retain most of their original functionality. Otherwise, we may render the system unusable for their particular use case. Since the user may no longer be in the default configuration, keeping track of the changes the user has made is critical for finding workarounds that maintain as much functionality as possible. Second, we want to avoid moving into configurations that we have previously determined are faulty, since this may cause new problems.

This particular failure has two possible workarounds. The value for *browser.startup.page* can be set to 1 or to 2. Either would avoid the failure, but because we want to keep as much functionality as possible for the user, we want to make the change that would have the smallest impact. Unfortunately, in this case we do not know which of the two reconfigurations is best. One option is to ask the user with a prompt which one is preferred, however that is not always possible and we will leave this as future

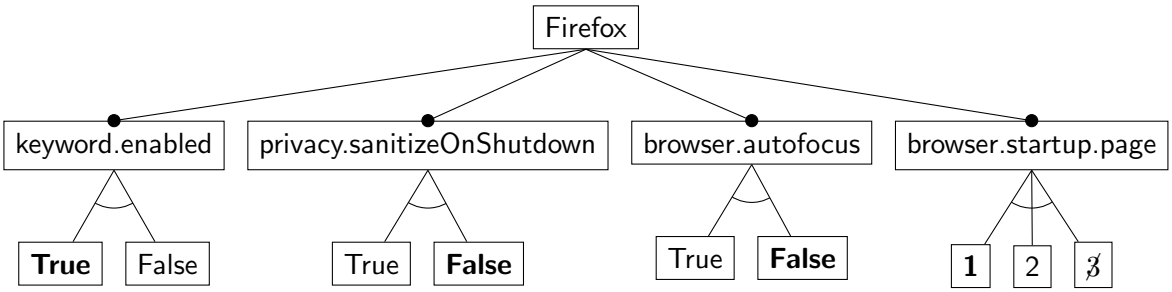

Figure 1.4: Updated Firefox Feature Model

work. In this thesis, if we can't determine which is better, we will break such a tie randomly. In this example we choose *browser.startup.page* set to 1. If, however, we could have also avoided this failure by changing the *keyword.enabled* to `False` and *privacy.sanitizeOnShutdown* to `True`, we would assume this would move use further away from the current functionality, because it is two option changes away, and thus would be rejected given that there are one-change possibilities.

Now that we know how to avoid the failure, we can also infer what caused it. Based on the failure and the found workaround we can assume that the reconfiguration option *browser.startup.page* set to 3 is at least partially to blame for the failure. Avoiding that feature in the future is one of the key aspects of the failure avoidance framework. This will be accomplished by implementing a guard [28]. Now the feature model has changed as shown in Figure 1.4. The value for *browser.startup.page* is no longer allowed to take on the value 3.

It is also possible that multiple configuration options must be changed to succeed with a workaround. For instance, if the workaround requires that *privacy.sanitizeOnShutdown* be set to `True` as well as *browser.startup.page* be set to 1 or 2, we now have a *two-hop* workaround. This means that the workaround is two reconfigurations different from the starting configuration (assuming each reconfiguration changes only a single option). This also means that the feature model will require multiple guards to pre-

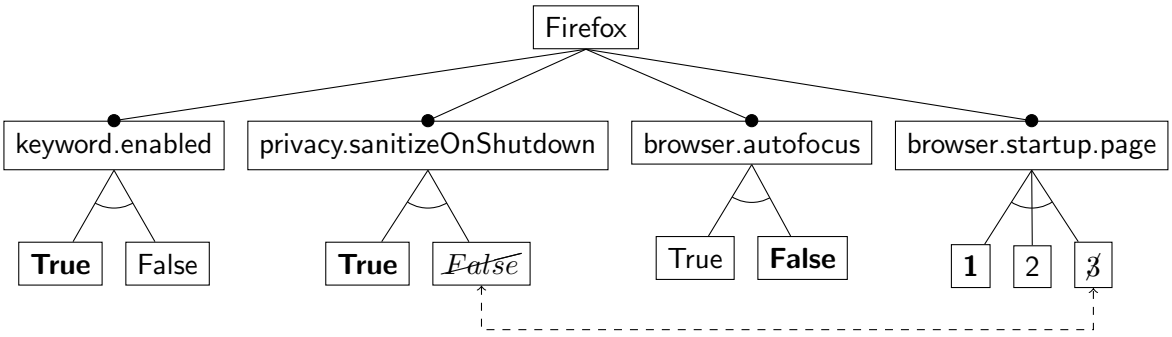

Figure 1.5: Updated Firefox Feature Model with Multiple Guards

vent the failure from reoccurring. This is represented in Figure 2.1, shown with an excludes dependency between the features.

1.2 Research Contributions

As seen in our motivating example, the potential exists for us to avoid failures in a system through prudent reconfigurations and to guard against future failures. In the work of Garvin et al. [25, 28], they developed an algorithm that searches for workarounds in a bounded brute force manner by searching all one-hop or two-hop reconfigurations for a workaround. However, since the configuration space grows exponentially large with the addition of features, this will not scale well and the bound must be small. Second, they evaluated their algorithms by re-creating the history of failures in the GCC compiler, but did not actually implement this idea in a real self-adaptive system.

This thesis extends the work of Garvin et al. in two significant ways. First, we evaluate a set of heuristic algorithms for finding workarounds that have the potential to be more efficient than the exhaustive one or two-hop algorithms. We evaluate these algorithms on the same GCC compiler failures and find that there are failures with no one or two-hop reconfigurations that our new algorithms can avoid. We also

see that our heuristic algorithms perform well with respect to effectiveness, finding workarounds for most of the failures that the brute force algorithms find. Second, we extend the Rainbow self-adaptive software system to work with our new utility function (failure avoidance) and adapt this to work on a set of Firefox clients. We used a set of real failures and determined that we can find workarounds and avoid all of these failures over time. We also found (and avoided) several native failures that we did not seed, but were able to reproduce once found.

In summary the contributions of this thesis are:

- An evaluation of a set of alternative heuristic algorithms for failure workarounds;
- A proposed self-adaptive framework for failure avoidance in configurable systems built on top of the Rainbow framework; and
- Experimentation with a set of Firefox clients and real faults showing the feasibility of this approach.

1.3 Overview of Thesis

The next chapter will cover background on the existing failure avoidance algorithms and Rainbow as well as discuss other related work. Chapter 3 presents our failure avoidance framework and the modifications that we made to Rainbow to realize this. Chapter 4 presents a case study to evaluate variations of the failure avoidance algorithm. Chapter 5 presents our case study on the Firefox self-adaptive implementation. Finally Chapter 6 concludes and presents future work.

Chapter 2

Background and Related Work

This chapter covers two main ideas that form the basis for this work. The first is the failure avoidance algorithm algorithm and the second is the Rainbow self-adaptive framework. We then present other related work.

2.1 Failure Avoidance Algorithm

Garvin et al. [27, 28] propose a set of algorithms for failure avoidance. This work is based on the notion of *feature locality* – the idea that multiple failures are likely to be manifested only under a small set of feature combinations, rather than be spread evenly through the configuration space. In this work they also confirm the existence of feature locality in the GCC compiler. In practice, feature locality means that if a failure can only be seen under a small set of configurations then there will exist a number of configurations that will avoid the failure and these may be relatively close to the failing ones (satisfying our first goal laid out in the introduction). In addition, bad configurations exist that can trigger multiple failures and by avoiding these configurations future failures can be avoided as well. This satisfies our second

goal presented in the introduction.

2.1.1 Algorithm Overview

The avoidance technique proposed can be divided into five steps. Step one reports information about a failure. Step two tries, by a brute force algorithm (discussed below), to find a similar configuration to the failing one, that passes the failing test case. Step three updates the system with a new passing configuration. Step four constructs a guard based on previously discovered good configurations. Step five then reconfigures the system. The core of this algorithm is that as configurations are found they are added to the guard database which can be used to circumvent failures as they are discovered. The original work was evaluated using three versions of the GNU GCC compiler [2] with faults from the GCC bug repository. The evaluated versions are 4.4.0, 4.4.1 and 4.4.2.

Algorithms 1 and 2 are taken from [27] and reproduced here. Algorithm 1 is the core of Step two. It works by first establishing the failure, generating a population of reconfigurations (line #4) that represent x-hop reconfiguration options (in the evaluation x was set to one and two). This is done by looking at the feature model, and changing each possible x configuration options from the staring configuration, one at a time. A loop (lines #5-13) then iterates over each reconfiguration, checks to make sure that the reconfiguration is valid, and runs the initial failing test case. If the results of the test are now a pass instead of a fail, then the reconfiguration can be stored as a workaround configuration. There is also a check at the end (line #15) to remove superset reconfigurations. This serves to eliminate redundant reconfigurations when the value of x is greater then one.

Algorithm 2 is the core of Step four. It looks at a given configuration c and

Algorithm 1 Algorithm for Analysis of Failures [27]

```
1: let t ← the reported test case
2: let c ← the reported configuration
3: let d ← the maximum number of feature groups to change at a time
4: let R ←the set of reconfigurations that affect between one and d feature groups;
5: for all r ∈ R do
6:     let c ′ ← c after applying reconfiguration r
7:     if t can be run under configuration c′ then
8:         if t passes under configuration c′ then
9:             note r as a known workaround
10:            note c′ as a passing workaround
11:        end if
12:    end if
13: end for
14: for all r ∈ R do
15:    if r is a possible or known workaround whose supersets in R are all either
       possible or known workarounds then then
16:        note r as a basis for generalization
17:    end if
18: end for
19: for all r ∈ R do
20:    if r is a strict superset of a a basis for generalization then
21:        Forget that r′ is a possible or known workaround
22:        Forget that r′ is a basis for generalization
23:    end if
24: end for
```

compares it to all found workarounds. If applying the workaround for some test case t changes c, then c is a potential dangerous configuration and should be rejected. If not c is safe as far as we know, and can be used.

To understand how the brute force algorithm works lets look at our earlier example. Starting with the user's current configuration a failure is encountered. To search for a workaround all $one - hop$ reconfigurations will be tested.

Figure 2.1 is the starting configuration for testing. We will now generate the set R based on all $one - hop$ reconfigurations from this configuration. There will be five of them, one for each possible reconfiguration. Note that there are 24 configurations

Algorithm 2 Algorithm for Guard on Configurations [27]

```
1: let c ← the reported configuration
2: let T ← the set of test cases with known workarounds
3: for all t ∈ T do
4:     if c is a passing configuration for t then
5:         continue with the next iteration of the loop on line 3
6:     end if
7:     for all basis for generalization r from t do
8:         let c′ ← c after applying reconfiguration r
9:         if c = c′ then
10:            continue with the next iteration of the loop on line 3
11:        end if
12:    end for
13:    reject c
14: end for
15: accept c
```

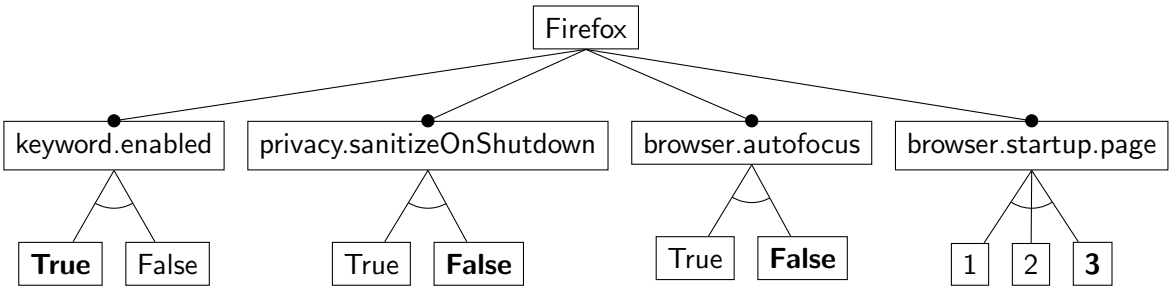

Figure 2.1: Firefox Feature Model Selected Configuration

in total, so while we call this brute-force it is only exhaustive with respect to the cardinality of the hop (i.e. one in this case).

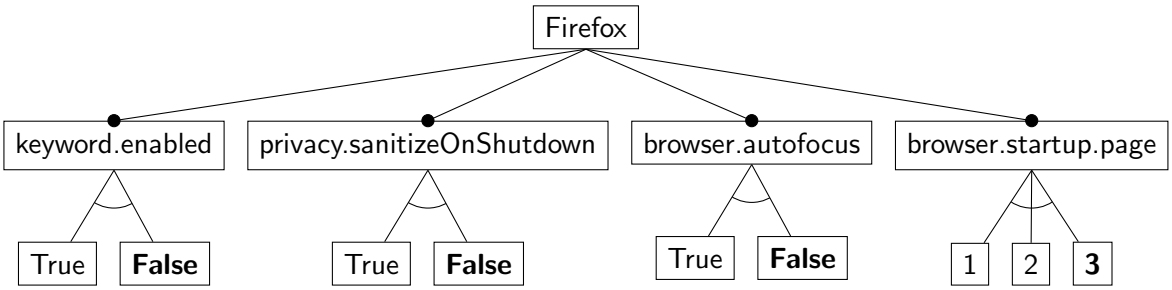

Figure 2.2: Reconfiguration Option 1

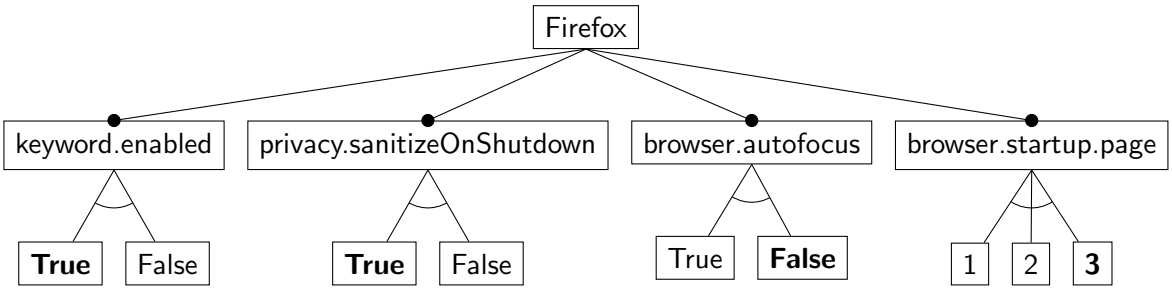

Figure 2.3: Reconfiguration Option 2

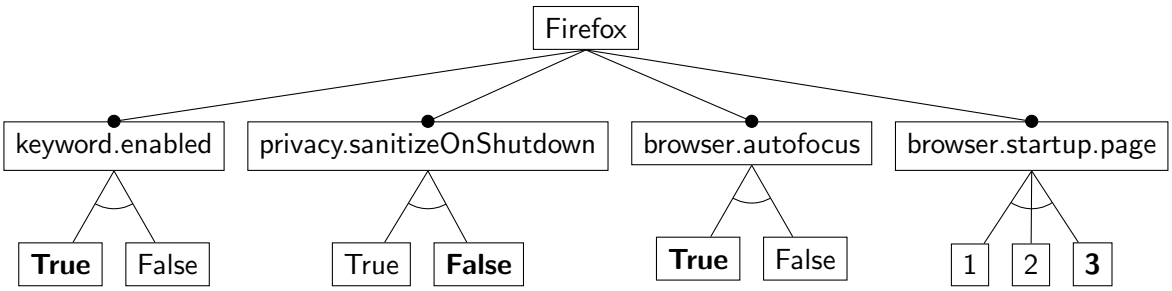

Figure 2.4: Reconfiguration Option 3

The configuration in Figure 2.2 changes *keyword.enabled* to `False`. The configuration in Figure 2.3 changes *privacy.sanitizeOnShutdown* to `True`. The configuration in Figure 2.4 changes *browser.autofocus* to `True`. The configuration in Figure 2.5 changes *browser.startup.page* to 1. Finally, the configuration in Figure 2.6 changes *browser.startup.page* to 2.

If we were looking at the same failure as we discussed earlier, then 3 of the tests

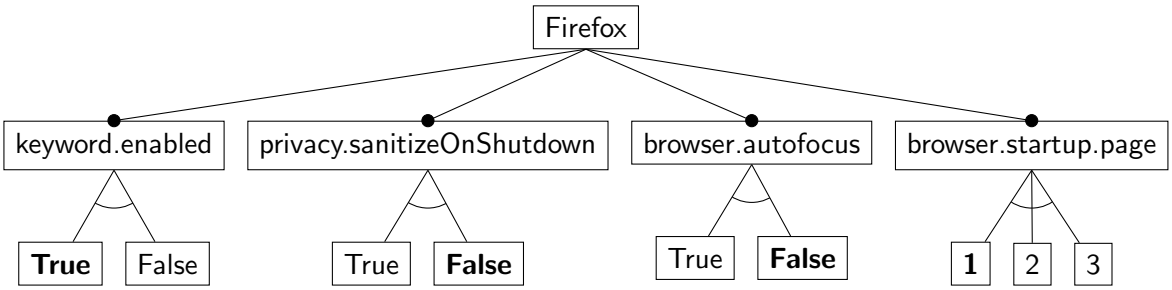

Figure 2.5: Reconfiguration Option 4

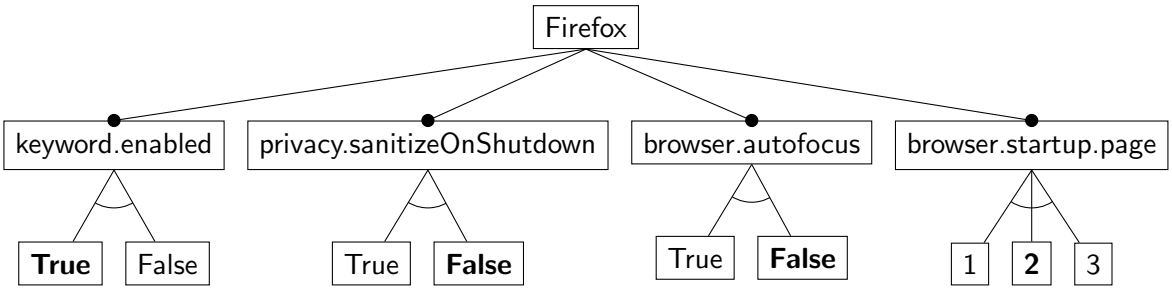

Figure 2.6: Reconfiguration Option 5

will still have failed, Figures 2.2, 2.3 and 2.4. However two of the tests, (2.5 and 2.6), would have passed. Because the failure was due to *browser.startup.page* set to 3, the 2 tests where *browser.startup.page* was not set to 3 will have passed and two workarounds will be discovered. We will then select one of these and add these for reconfiguration and add both workarounds and guards to our set.

2.2 Rainbow Self-Adaptive Framework

Rainbow [7, 10, 23, 24], developed by S.W. Chen, D. Garlan and B. Schmerl, is a self-adaptive framework designed to answer not only the what of automatic system adaptation, but also the when. By taking advantage of the architecture of a system and focusing on a select number of critical properties, a system can be monitored during runtime to allow for effective adaption for a number of possible issues. The

existing rainbow implementation, provides a rainbow server and a client, `znn.com`, a web browser that opens the fictional website.

Rainbow is built with three core objectives. It is meant to be general so that many architectural styles can be used and Rainbow isn't limited to what systems can be adapted. It should be cost-effective, so that using the self-adaptive capabilities don't compromise the usefulness of the system. Finally the self adaptation should be transparent so that multiple objectives can be measured and compared. We tried to maintain these objectives when extending it for our purpose.

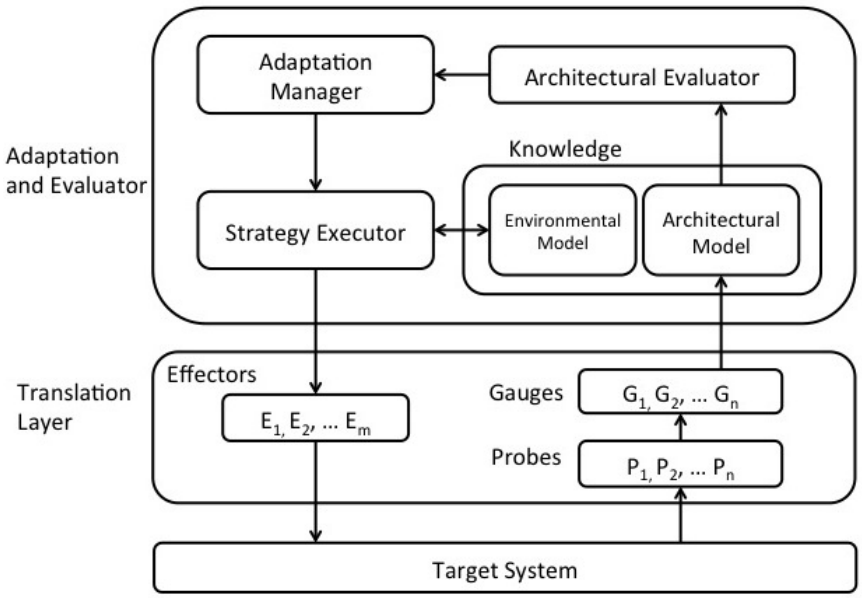

Figure 2.7: Rainbow Overview

2.2.1 Overview

An overview of the Rainbow architecture is shown in Figure 2.7. The top layer (the Adaptation and Evaluator Layer), contains information about the architecture of the system including a model of both the architecture and environment, evaluators for the reconfiguration strategies and execution managers. At the lowest level is the system

that adapts (called the Target System). In the middle layer, sit a set of *probes* and *gauges* and *effectors*. Effectors interface with the target system to make changes. The effectors and strategies are both written in the *Stitch* script language.

Rainbow uses the probes and gauges to monitor the target system. Depending on the values obtained from these, Rainbow will use the strategies to determine what reconfigurations to make. These are then implemented by the effectors. These are meant to provide an automated way to perform what would otherwise be human tasks. For instance, if a person monitoring a website sees that the latency has increased then they could bring additional servers online to try and bring the latency down. Rainbow will take the human out of this example and instead make the necessary changes based on the adaptation strategy which should also tell it to bring additional servers online [9].

Rainbow follows the IBM autonomic MAPE architecture [22]. This means that there are 4+1 phases that Rainbow will use.

- Monitoring − This phase handles collecting the necessary system data. There are multiple ways to accomplish this, such as instrumenting source code or reading system logs. Because the monitor is constantly running the overhead must be kept as low as possible. This means that the data collected will be simple to collect, and require very little or no logic to decipher.

- Detection − This phase interprets the data collected through monitoring. In addition the phase will determine if it is possible to improve the system using any of the available adaptation strategies.

- Decision − This phase chooses the best adaptation strategy to improve the system. It might also be used to determine what is causing the system to underperform.

- Action − This phase executes the chosen strategy, and then checks to make sure that the system is performing as expected. This phase might need to deal with system level errors in case the strategy fails, or the desired result is not obtained.

- Knowledge − This component manages the data shared between the separate phases to facilitate the adaptation.

The Rainbow Framework uses a number of components to provide the monitoring, detection, decision, action, and knowledge phases. Now we will look at how each of these components represent the phases needed for self adaptation.

2.2.1.1 Translation Layer: Monitoring and Action

Correspondence between the target system and the architecture requires a component that connects the properties from the architecture model to the target system. The translation component provides this capability taking abstract data from the target system, and interpreting it into the appropriate property in the model. In addition, changes may be made to the target system by Rainbow in order to adapt the system, and this requires a bridge between Rainbow and the target system.

Getting information from the system is accomplished through Rainbow's probes and gauges, which represent monitoring. The probes are used to measure some aspect of the system, while the gauges interpret the data the probe is looking at, and updates the architecture model accordingly. Each probe is mapped to a specific gauge, which means that multiple types of probes can be employed. For instance a probe can check for the existence of some file, or read data out of a log. It is also possible to use different gauges to interpret the data in different ways, such as a gauge which just reports the value the probe gives it, versus one that reports the average value of the

probe it is connected to. This one to many relationship is why the probes and gauges are separate.

Changing the target system is done using effectors, which represent part of the action phase. These are open ended mechanisms that range in potential and use. An effector can be as simple as making a system call, or it could be more complicated, such as running a predefined script. Without effectors Rainbow would have no way of changing the target system, and therefore no way of adapting it to changes.

2.2.1.2 Evaluator and Adaptation Layer: Detect and Decide

The rainbow architecture evaluator is run every time a model property changes. If a constraint is violated by the change, then the adaptation manager is notified to begin its search for adaptation. The evaluator defined in Rainbow as a constraint that can be added to the architecture model and checked after each change. This takes care of the Detect phase.

The adaptation manager handles the decision phase for Rainbow. Once activated, the adaptation manager will use the models to select a strategy that best solves the current problem. Choosing the best strategy involves checking multiple strategies and evaluating the potential results of choosing that strategy. Being able to define multiple strategies and let the adaptation manager choose the best one based on a number of system concerns is one of the core advantages of using Rainbow [23]. Each strategy is made up of one or more tactics and each tactic is a sequence of commands. These are defined using a stitch script, and can be tailored for a variety of situations, such as aggressive strategies that might overcompensate but guaranteed to solve the issue, verses a more timid strategy that is less intrusive to the system, but might not always solve the issue.

In addition, a strategy executer is employed which hooks to defined effectors to

carry out the adaptation. This is the second part of the action phase, and deals with handling issues that might arise, such as errors from the effectors or failure of a tactic to resolve the issue.

2.2.1.3 Knowlege Component: Models

There are two models used in the Rainbow framework. The first is the architecture model, which represents the state that the system is in. This includes information such as the current properties and constraints on the target system. The architectural model is updated to reflect the current values of properties as the system is running. The other model used is the environment model, which represents information about the system. This includes information such as the current execution environment, and resources available. While the architecture model is used to determine if something can be improved, the environment model allows Rainbow to know what adaptation strategies might perform better.

A model manager is used to update and access the architecture model. This manager is also used to deploy the translation component and is queried by the adaptation manager to know what strategies and techniques to use. The model manager represents the Knowledge process of the Rainbow framework, and is essentially the glue that keeps the self adaptation running.

2.2.2 Existing Version of Rainbow

The primary Rainbow example provided is a mock news website to showcase Rainbow's ability to adapt to a changing system. As latency on a web page increases, multiple strategies can be deployed to improve the user experience. First, additional servers can be brought online to handle the increased load. If that fails the content

on the web page could be decreased. For example pictures are removed, which means less information to load, and an improved wait time for users. Even though the functionality is decreased because the content has been lessened, the alternative is no one being able to access the page which is worse.

2.3 Related Work

We now present some of the other related work broken on self-adaptation, finding failure workarounds and on configuration-aware testing.

2.3.1 Self-adaptation

There has been a large amount of work done recently on the topic of self-adaptive systems. The concept of a self-adaptive system is related to the idea of autonomic computing [22]. As software continues to become more and more complicated the idea of a system that can regulate itself effectively and efficiently becomes increasingly worth the investment. Self-adaptive software has been used on software product lines [50], which transitions well into our work on configurations.

Some of the early work on self-adaptive software deals with systems that can heal themselves. For instance the work of Dashofy et al. [16] looks at targeting event based software architectures and requires a large infrastructure to define and implement the repairs. This work shares some similarities to Rainbow.

Some of the work in self-adaptive systems has focused on improving a particular aspect. The work of Esfahani et al. [21] deals with weighing the pros and cons of an adaptation strategy and how to choose one. The biggest issue is dealing with the uncertainty of what a decision is supposed to do vs what actually happens. The

work of Ebnenasir et al. [19] looks at updating the program once a failure has been encountered.

There has also been work done meant to support self-adaptive systems. This includes the work of Georgas et al [29] which looks at the architectural configurations for a system and can reconfigure to avoid bad states. This is fully decoupled from the self-adaptive adaptation manager. In terms of system validation Zhang et al. [59] worked on splitting the analysis and validation of self-adaptive software into adaptive and non-adaptive to better understand self-adaptive software.

There has also been work on building complete self-adaptive systems such as FUSION by Elkhodary et al. [20] FUSION makes system adaptations based on a model constructed from features. While these features can be similar to the features we are using, they are adapting based on quality of service attributes. By abstracting the technique from a specific architecture and instead focusing on features the system can be adapted to handle unexpected conditions. Other work by Perkins et al. [42] observes the system to determine standard behavior, detects errors when they occur, creates and deploys a patch that enforces a failure to be true and monitors the deployed patches and updates them when necessary.

Finally, Salehie et al. [49] provides an overview of a number of self-adaptive systems. This survey of current trends in self-adaptive software highlighting difficulties and areas for improvement. Rainbow was featured in this analysis and compared well to other systems. Rainbow was the only self adaptive framework that provided self-configuration, self-healing and self-optimization properties, as well as providing explicit features for extensive monitoring and basic features for detection, decision and acting. Even though some frameworks such as ML-IDS and Self-Adaptive can outperform Rainbow in regards to specific features, Rainbow provides solid overall functionality and this makes it ideal as a base for our failure avoidance framework.

2.3.2 Failure Workarounds

Workarounds can be used to supplement standard software testing, as understanding how to avoid a failure can give great insight into why the failure exists, as well as providing users a method of avoiding the failure until a fix is found. There has been work done on adapting software to avoid failures using standard APIs [17] or using alternative but equivalent execution sequences [31].

Other related work is that of automatically finding patches using genetic programming from the work of Weimer et al. [54]. Their approach finds faults in systems using standard tests, with no formal specifications required. An extended form of genetic programing is used to find a workaround for the failure. The workaround is then minimized using delta debugging similar to our technique.

ARMOR is a tool designed by Carzaniga et al [5], [4], [3]. The goal of ARMOR is to make applications more resilient to failures during runtime. This is done by finding workarounds for failures, but instead of the workarounds being based on reconfigurations, their workarounds are based on alternate library calls. This technique is similar to ours in its goal and the requirements to find a workaround. While our technique needs a feature model for the system, they require a list of alternative library calls. Both techniques need the failure to be replicable in order to determine that the workaround is really a workaround.

The work by Cleve et al. [12] examines the state differences between a passing and failing run of a program. By focusing on the relevant data at the time of the failure the cause of the failure can be identified. This leads to a better understanding of the failure and how to replicate it. In addition the work of DiGiuseppe et al. [18] further looks at faults and how they interact within a program. Specifically the faults which obscure other faults, and would imply that workarounds can lead to new failures.

Workarounds can be concentrated in the system shown by the work of Kim et al. [33]. This is done by caching the location of a fault in a system to find areas to focus on further testing and validation. Results show a large number of faults are located in a minority of the source code files. This can be related to the concept of feature locality, as if failures are concentrated in source code it would follow that a feature which includes the buggy source code will be responsible for multiple failures.

2.3.3 Configuration-Aware Testing

Testing the right configurations is both difficult and critical for large highly-configurable systems. An exponential increase in the number of configurations with features, means a large increase in the number of ways something can go wrong. Because of this, a lot of work has been done on how to effectively test the configuration space [14, 25, 34, 44, 48, 55, 55]. There is also a body of work on this topic in the software product line testing community [1].

Part of being able to perform configuration-aware testing comes from the idea of feature testing [11]. In their work, Classen et al. examine what makes a feature and how to define feature interaction. Further work by Garvin and Cohen [25] more formally defines the notion of feature interaction for configurable systems.

Many sampling techniques have been used for selecting a subset of configurations for testing. The use of covering arrays to achieve good samples for testing configurable systems [56], [46] and software product lines [43] is common. For instance, the work of Lochau et al. [36] use pairwise combinations to efficiently test software product lines. This style of testing involves covering arrays which we use in some of our new failure avoidance algorithms. These will be discussed in more detail in Chapter 4. Since software product lines are a type of configurable system, the work which focuses on

feature models of SPLs such as [15], [30] can be used to create better feature models of a configuration space.

Another way to improve configuration testing is to introduce prioritization. The wok of Sriknanth et al. [51] looks at prioritizing what configurations to test by examining setup time and relevance to the feature being tested as well as previous failures. This is also seen the work by Qu et al. [45].

One aspect of configuration testing is validating a configuration [52] and understanding the configuration space. Reisner et al. [47] used symbolic execution of configurable software systems to show a much smaller configuration space possible then previously thought.

Chapter 3

Failure Avoidance Framework

In this chapter we present our new failure avoidance framework that we built upon the Rainbow framework. A number of changes were necessary to Rainbow which we detail below. The framework is shown in Figure 3.1. In this figure we see a set of clients running. A client is representative of a user of a system (such as Firefox) which means that each client will have their own build of the system. Each client will have an error probe and an error gauge so that when a failure occurs on the client the information (current configuration, failing test case) is sent to the failure avoidance algorithm to find a workaround. The workarounds use the system model (in our case the firefox feature model and the history of past failures). Each client will also keep track of the workarounds already discovered which allows them to implement a guard to prevent seeing the same failure twice. When a workaround is found a change is then made to the client that failed.

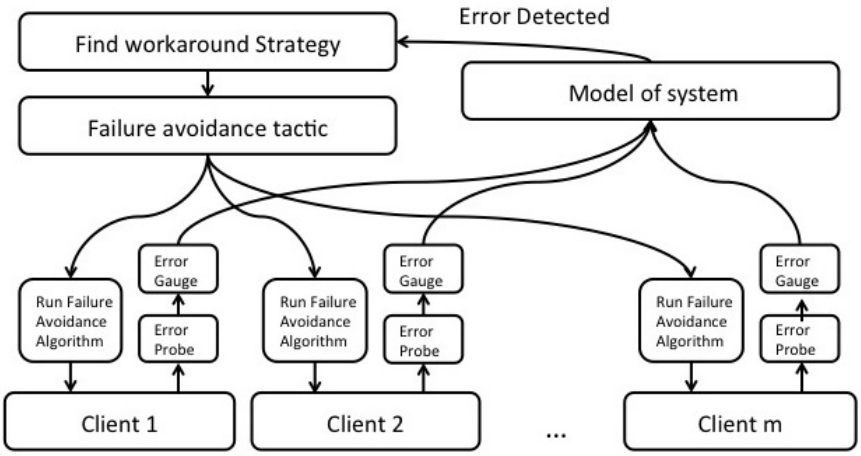

Figure 3.1: Failure Avoidance Framework

3.1 Architecture of System

The architecture is designed for use as a distributed system with a number of clients running the system in parallel. A single central master is responsible for maintaining information about the existing guards and is responsible for running the failure avoidance algorithm when needed. We felt that this distributed client model would be more representative of the real world. Each of the clients will be monitored for failures, and in a real world these failures will be replicated by creating a test case of the clients behavior and actions prior to the failure. In our simulation we will be using mozmill tests that represent client behavior prior to a failure. The recreation of a failure in a real world setting is saved for future work.

3.1.1 Feature Model

The system model that is monitored/used in this framework is the feature model. The feature model is made up of properties of the system that will be monitored. Because we want to adapt to failures the failure avoidance framework needs a failure property.

```
$Property error : int << HIGH : int = 1; LOW : int = 0; default : int = 0; >> ;$
$ rule primaryConstraint = invariant self.error <= 0;$
```

Figure 3.2: Example Property and Constraint

It is called *error* and is an integer with a high value of 1 and a low and default value of 0. This represents no error for 0, and an error detected for 1. To determine if adaptation is necessary a constraint needs to be violated for some property. In our system a new constraint rule is also added that says the error value should not exceed 0. Our initial version of the framework makes this property binary, but it can be expanded to have multiple values. We leave that as future work. For instance instead of the current 0 or 1, it could represent a scale from 0 to 5 where 5 is a critical failure and requires immediate attention. An example of the new error property and constraint is shown in figure 3.2.

3.2 Modifications for Monitoring

The first part of avoiding failures is simply detecting when a failure has occurred. We describe the necessary changes for monitoring we made here.

3.2.1 Failure Probes

A new probe will be used to detect when a failure has occurred. This probe is defined in the `errorProbe.pl` file and determines if a failure has occurred by checking for the presence of an *error flag* file. The use of a flag file was chosen because it allows us to remove the Rainbow architecture from the failure detection to determine if an error has occurred. In this way, it does not matter how an error was detected, as long as an `error.true` file is created in the appropriate location. We can detect a

failure in this way by running test cases, or by user reports in theory. In addition the `probe.yml` file is extended to work with the probe.

Each client has its own probe, which means that each client also has its won `probe.yml` file. This allows for customization of each probe, so that each one can be checking for the file in a different location. For instance, client 1 error probe checks for the file in the client 1 directory, and so on.

3.2.2 Failure Gauges

Because a new probe was added, at least one new gauge must be added as well. To do this a `new errorGauge` is created and added to the Rainbow distribution (this is contained in the main jar file). The new gauge simply collects the report from the probe and updates the model with the result. In addition the `gauge.yml` file in the model directory must be updated as well for each of the gauge instances.

Unlike with the probes, the gauges must be declared in the Master and in each client. In order to get Rainbow to detect that a gauge should be running the master must have a copy of each gauge instance. This means for x clients there will be x error gauge instances on the Master, and 1 on each client.

3.2.3 Effectors

Interacting with the model is done using the failure avoidance algorithm. Calling the effector will launch the algorithm giving it the client information (what test failed and the current configuration at the time of the failure). The algorithm will then attempt to find a workaround for the failure and update the client with the results. Even if no workaround is found the effector must remove the `error.true` file, otherwise Rainbow will continually try to run different strategies to satisfy the primary constraint. This

is a change that is needed because our algorithms take some time to perform the analyze phase.

One of the major benefits of the effector is that the results of searching for a workaround, if positive, can be shared across all clients. This means that if client one encounters an error, and finds a workaround, client two will gain knowledge of the workaround and client two's guard is improved. Sharing information allows for failure avoidance without needing a user to actually encounter the failure themselves. This is done to allow the system to more quickly reach a steady state, which is where no new failures are encountered.

3.2.4 New Stitch Scripts

Three major changes were made to Rainbow's adaptation scripts. First, a new value vector was added to the utility function which allows for tracking the average value of the error property. A new tactic was designed, the *failure avoidance tactic*, which calls the effector that searches for a workaround using a failure avoidance algorithm. In addition the tactic was added to the utility function where it is specified to improve the value of the error property. This is so that the Rainbow adaptation manager knows that this tactic can be applied to fix a violation of the primary constraint. Finally a strategy, *Search for Workaround*, was added. Search for workaround is called when the primary constraint is violated and it runs the new Failure avoidance tactic.

3.2.5 Main Algorithm

Up until this point one failure avoidance algorithm has been described (and used) to find workarounds, however additional algorithms could be added which would

improve the adaptation process by allowing multiple tactics. For instance if time is of the essence, a faster algorithm that is not as thorough vs. a slower algorithm which will find more workarounds to improve the guard. Algorithm 3 shows the high-level algorithm that is used in this framework.

We start with a set of target client systems and a set of failure avoidance algorithms. We also have an architecture which includes the feature model. Our initial algorithm assumes that a single tactic will be used while running (we leave the use of dynamic tactic selection as future work). While the system is being monitored, if a failure is discovered then the client with the failure is locked and the failure avoidance algorithm runs. When a workaround is found the client is updated to its new configuration. In addition, the workaround information is also deployed to all clients so that they can update their guard.

Algorithm 3 Algorithm for Failure Avoidance Framework

```
1: let C ← the set of target client systems
2: let A ← the set of failure avoidance algorithms
3: let M ← the architectural model of the framework
4: let t ← the chosen failure avoidance tactic
5: for all c_i ∈ C do
6:     Instantiate an error probe e_i and error gauge g_i on c_i, and map them together
7: end for
8: while System is being monitored do
9:     if e_i detects a failure then
10:        c_i is locked
11:        g_i updates M with the failure
12:        Error Constraint in M is violated, the Search for workaround strategy is
       called
13:        Failure avoidance tactic t selects a failure avoidance algorithm a ∈ A
14:        a replicates the failure from c_i, attempts to find a workaround
15:        if a discovers a workaround w then
16:            w is deployed to all c∈ C so that each c can update its guard
17:        end if
18:        c_i is updated with result from a, c_i is unlocked
19:    end if
20: end while
```

Chapter 4

Heuristic Failure Avoidance Algorithms

The failure avoidance framework (see Figure 3.1) provides the flexibility to include different failure workaround algorithms. In initial work by Garvin et al. [27, 28], the authors propose an algorithm for finding workarounds that exhaustively searches one feature change from the current configuration (called from now on the *one-hop algorithm*), i.e. each feature is switched on/off in turn to determine if the change will lead to a non-failing test case. In that work, they also performed an exhaustive two-hop search (*two-hop algorithm*) without finding any additional workarounds. However, their algorithm can still miss potential workarounds; any beyond one or two-hops will not be found. In this chapter we propose several additional workaround algorithms, ones that are stochastic and will explore beyond the two-hop boundary. We then evaluate these on the same open source program used in [27], GCC.

Algorithm 4 shows the modified failure avoidance algorithm as was presented in [27]. The only change we make to that algorithm, is that we change the set of reconfigurations to a population of possible workaround configurations. In this

Algorithm 4 Algorithm for Finding Workarounds

```
1: let t ← the reported test case
2: let c ← the reported configuration
3: let R ←the set of given possible workaround configurations
4: for all r ∈ R do
5:     if t can be run under configuration r then
6:         if t passes under configuration r then
7:             let r′ ← the minimized configuration subset of r where t can be run
   under configuration r′ and t passes under configuration r′
8:             note the set of reconfigurations to reach r' from c as a known
   workaround
9:             note r′ as a passing workaround
10:        end if
11:    end if
12: end for
13: for all r ∈ R do
14:    if r is a possible or known workaround whose supersets in R are all either
   possible or known workarounds then then
15:        note r as a basis for generalization
16:    end if
17: end for
18: for all r ∈ R do
19:    if r is a strict superset of a a basis for generalization then
20:        Forget that r′ is a possible or known workaround
21:        Forget that r′ is a basis for generalization
22:    end if
23: end for
```

algorithm lines #3 and 4 from the original algorithm are consolidated into line #3. By doing this, it means that the manner of finding the population is not fixed, so any of the algorithms we propose can simply be be plugged into this line. In addition, once a configuration has been found to pass a failing test, it must be minimized before the result can be added to the pool of known workarounds (line #7). We discuss the need and method for minimization below. The rest of the algorithm mirrors the orignal algorithm from Chapter 3.

4.1 Limitations of the One and Two-hop Algorithms

The one and two-hop algorithms both use a brute force approach to explore the immediate population space around a failing configuration. Each workaround found will be one or two option changes away from the failing configuration, and each workaround will contain exactly one reconfiguration. This has the advantage of keeping the workarounds close to the starting configuration which minimizes the chance that the workaround will impact the intended functionality of the user.

Although the two-hop algorithm may find more workarounds, the result is an increase in time and cost. The problem is that finding workarounds with this method constitutes an exponential growth with respect to the number of features in the feature model. As systems continue to grow larger, and feature models therefore increase in size, anything more than a one-hop algorithm becomes infeasible. In the study by Garvin et al. [27] although some new two-hop workarounds were found for failures, these were failures that already had a one-hop workaround, so they were of minimal value. In this case, the additional cost of the two-hop may not be justifiable. We explore this issue further in our case study.

4.2 Heuristically Exploring the Population Space

Since one of the main principles of the one or two-hop algorithm is that if a workaround exists it will be close to the starting configuration, this means that the number of reconfigurations necessary to reach a safe state is small, and it means that the potential functional impact of the workaround is kept small as well. We also know that multiple workarounds might exists for the same failure, and these workarounds will

often be related. For instance, if we examine the compiler, GCC, it uses a flag, *-W* which controls warnings. For all circumstances, where the use of the *-W* flag serves as a workaround, the *-Wall flag* will work as well and we would like to include these in our workaround set. There must be some difference between these features even though they can both be used to avoid the same failure. Because the functional impact can change on a feature by feature basis finding all of the possible workarounds is necessary to ensure that the minimal functional change is achieved. We believe that a search for reconfigurations further away from our starting point is necessary to find some of these additional workarounds.

In addition, in the work of Kuhn et al. [34], the failures may be due to more than two or three configuration options, albeit less often than due to one or two configuration options. This leads us to believe that there are going to be multiple ways to avoid a failure using a one-hop workaround, and it also suggests that some workarounds may not be found without only a single reconfiguration. Since the x-hop approach approach will clearly scale poorly for large configuration spaces, we need a heuristic method to search for new workarounds. In addition, the time it takes to find even a one-hop reconfiguration workaround is bounded by the number of reconfigurations. Could this number be improved upon with a different approach? These questions prompted further research that led to the following algorithms, all of which are heuristic.

4.2.1 Genetic Algorithm

Our first attempt to improve upon the x-hop approach is a genetic algorithm. A genetic algorithm (GA) is a metaheuristic search algorithm, that can be used to find solutions to optimization problems. The two key parts of a GA are the representation

of the candidate solutions, and a fitness function that is used to evaluate those candidate solutions. The candidate solutions can be selected randomly or by seeding, and they are ordered based on the fitness function. The highly ranked ordered candidate solutions are then mixed using a crossover pattern to speed convergence, and then partially mutated to introduce diversity. The new candidates are ordered again, and the cycle repeats until either a set number of generations are concluded, or an optimal solution is found. GA's have been used when dealing with many aspects of software engineering testing including test suite generation and because of this it seemed like a fair conclusion to try and adapt a GA to represent a configuration.

To apply a GA to this problem we needed a fitness function. The original fitness function we try is binary where 0 represents that the test with the candidate configuration passes, and 1 represents that the test run with the candidate configuration fails. After some initial testing the fitness function was enhanced to improve the quality of workarounds found. The new fitness function uses a distance metric. The distance represents the number of reconfigurations necessary to reach the potential workaround configuration from the starting configuration. There is also an additional value added if the test run is not a workaround. For our experiments the additional value is the total number of configuration options from the feature model. The idea here is to minimize the workarounds found and try to converge to a workaround that is much closer to the starting configuration. The algorithm for finding the fitness is shown in Algorithm 5. This fitness function will hopefully prune out as many unnecessary configuration changes as possible and help minimize the workarounds found.

To order the configurations for selection, first it is ensured that the candidates do not violate any of the existing constraints on the configuration. Then each candidate is run through a driver that returns whether the test passes or fails. If the test

Algorithm 5 Algorithm for Fitness Function

```
1: let t ← the reported test case
2: let c ← the potential workaround configuration
3: let s ← the starting configuration
4: let d ← the number of reconfigurations to reach from c from s (distance)
5: if t can be run under configuration c then
6:     if t passes under configuration c then
7:         return d
8:     end if
9: end if
10: return d + number of configurations in s
```

passes with the reconfiguration, then the passing configuration is compared with the failing one. If an option is different it is added to the reconfiguration set. This set of reconfigurations is then added to the set of workarounds (assuming it is not already in this set).

Two problems emerged from the use of the genetic algorithm for finding workarounds. First, the GA tended to converge towards a single workaround, which was not always the smallest workaround possible for that bug. Second, the GA seemed to converge closer to the starting configuration more quickly as the mutation rate was increased. This was partly due to the limitations of the fitness function, and partly influenced by the nature of the problem. A configuration that serves as a workaround may still be a workaround with the addition of superfluous reconfigurations. The problems with the GA converging to one workaround might be solved with the addition of multiple runs of the GA starting from scratch each time and an extension to the fitness function to avoid previously found workarounds. However, since we were seeing better results as our mutation got higher (and hence we were adding more randomness) we decided to first try a completely random algorithm. We leave the use of a genetic algorithm as future work.

There were some workarounds found for failures that did not previously have a

workaround. This would indicate the existence of at least a three-hop workaround and suggests that examining the larger search space may be profitable.

4.2.2 Random Search

The genetic algorithm was slightly modified so that a random algorithm could be tried. To come up with the starting population for the genetic algorithm a number of configurations are randomly generated. The random algorithm keeps this step in tact, but instead instead of crossover and mutation, the randomly generated starting configuration is simply checked to see if it serves as a workaround. This means that for a feature model with 100 different configurations for the same cost as the one-hop algorithm you can check 100 completely random configurations and explore a much larger size of the configuration space. The downside of this approach, of course, is that it lacks any intelligence in configuration selection.

The only check that is needed after generating a random configuration is to make sure that the configuration does not violate any of the constraints of the feature model. If it does a new random configuration can be generated. This is to ensure that none of the tests are wasted by checking a configuration that could never serve as a workaround

The overall idea is that if each configuration option is a switch on a giant switchboard. A random configuration amounts to looking at each switch and giving an equal chance to pick each option. This works well for finding a workaround because in many cases as long as the feature that exposes the fault is changed the configuration will serve as a workaround. For example, if a workaround exists as long as $-Wall$ is chosen, and random configuration has a 50/50 chance of selecting $-Wall$ because it can either be included or not included, then each random configuration

has a 50/50 chance of being a workaround. Increasing the number of features necessary for a workaround will decrease the odds of a random configuration from being a workaround.

The only thing that affects this model is the addition of masking. Certain configuration options might have an affect on other configuration options. For instance the compiler GCC has two configuration options, $-fschedule-insns$ and $-fschedule-insns2$. If a workaround requires that the option $-fschedule-insns2$ be set and $-fschedule-insns2$ requires that $-fschedule-insns$ be set then consider what happens when configuration option $-fschedule-insns$ is not selected. The presence of $-fschedule-insns2$ will be masked because the necessary option $-fschedule-insns$ was not set. In this way even though only one reconfiguration is necessary for a workaround, the presence of another reconfiguration can essentially remove the first one.

The problem with this completely random approach is that we now need to determine what the *necessary* reconfiguration(s) for a workaround are. A workaround that involves changing half of the configurations for a system may work, but it might also disrupt the functionality so much that it is of no use to the user. Much like with the GA, we want to reduce the reconfigurations to as close to the starting configuration as possible. Because there is no fitness function as in the GA to drive us towards better reconfigurations, a different method was needed. For this a *minimizer* was implemented.

4.2.3 Minimizer for Workarounds

The minimizer works using a delta debugging technique [58] for finding the smallest number of reconfigurations necessary for a workaround. The algorithm will check the given configuration, C to make sure it is a workaround. This means that for our

purposes a failing test case for the delta debugging algorithm is actually one that passes. In effect, the smallest failing inducing input is the fewest reconfigurations necessary for a workaround. In other words removing any of the reconfigurations will result in the test to fail, and the workaround will no longer be valid.

To find the minimal workaround binary search is used. Given the starting configuration S and the starting workaround configuration C. C will be split into partitions where some of the reconfigurations are from C and the rest are from S. The partitions will then be checked to see if they serve as a workaround. If a partition c is a workaround, then C will be reduced to c, and the process will restart. In this way the algorithm will systematically reduce the number of reconfigurations in C that are different from S until a minimal C is found.

For instance C will be split into two partitions c_1 and c_2. if c_1 is a workaround then reduce C to c_1 and start again. If c_2 is a workaround then reduce C to c_2 and start again. If neither serve as a workaround then split C into a small c_i and a large c_i. test all the small c_is first, and then the large c_is. If at any point c_i serves as a workaround then reduce C to c_i and start over. If none of the c_is are a workaround then double the number of partitions generated. Repeat this until the number of partitions is greater then the number of reconfigurations different from the starting configuration. This means that the current C is the minimal workaround. In the worst case the cost is $3|\mathrm{c}| + |\mathrm{c}|^2$.

Using the minimizer a randomly generated configuration that serves as a workaround can be reduced to the minimal subset of reconfigurations necessary to avoid the failure. In this way the problem that the GA had with workarounds that are different only because of superfluous workarounds is avoided.

Lets revisit our example from the introduction to see how random with a minimizer will work. Again, we start with a selected configuration in figure 4.1.

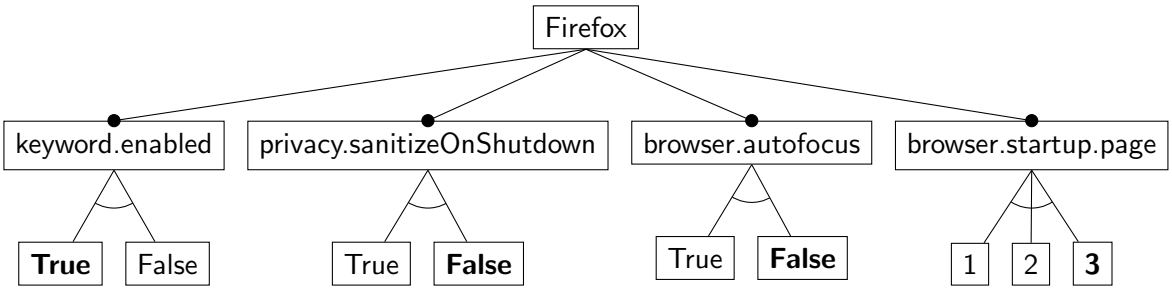

Figure 4.1: Firefox Feature Model Selected Configuration

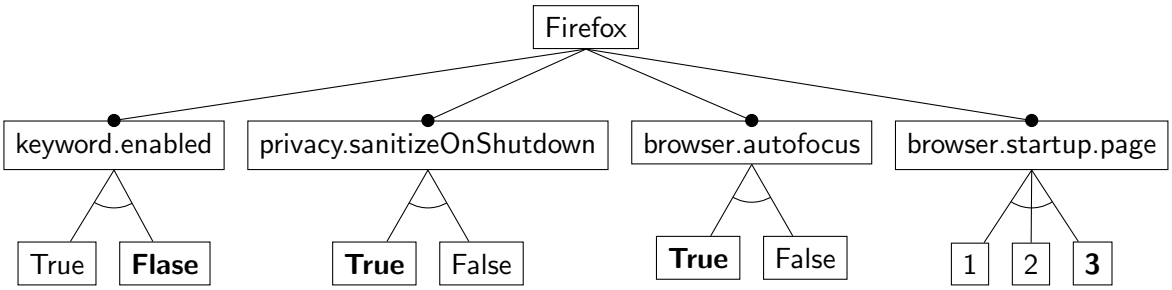

Figure 4.2: Randomly Generated Configuration 1

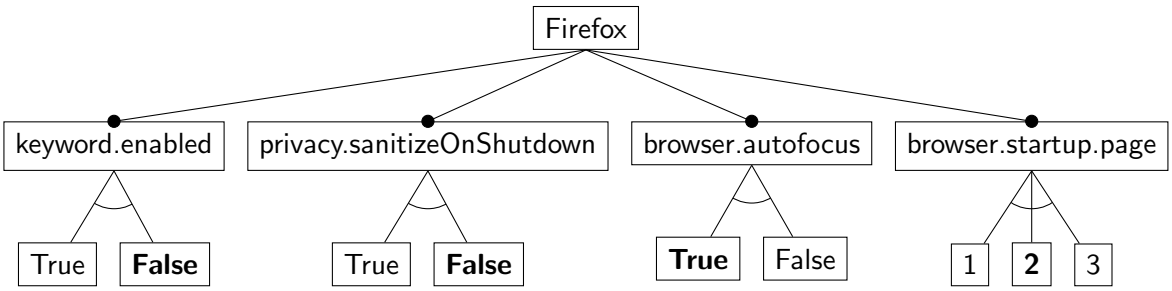

Figure 4.3: Randomly Generated Configuration 2

Now we encounter a failure. The one-hop algorithm algorithm would generate a set of possible workaround configurations by making one change on each configuration in the set. The random algorithm is just given a size for the set and then randomly generates a set of possible workaround configurations. For this example lets say that 2 randomly generated possible workarounds are generated. One in figure 4.2, and the other in figure 4.3

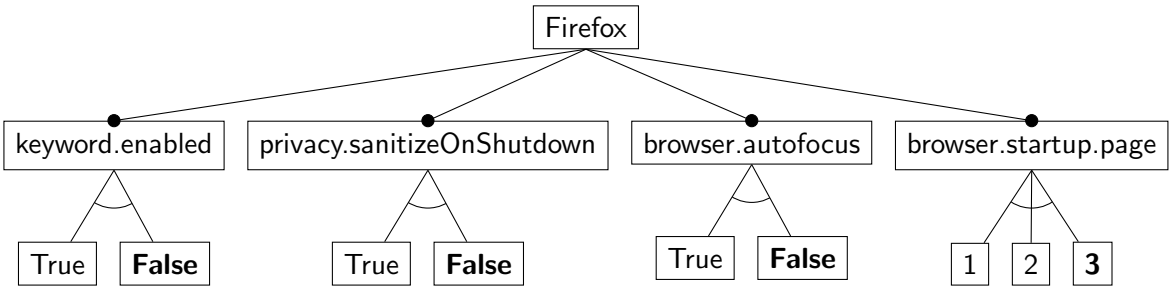

Figure 4.4: Minimized Configuration 1

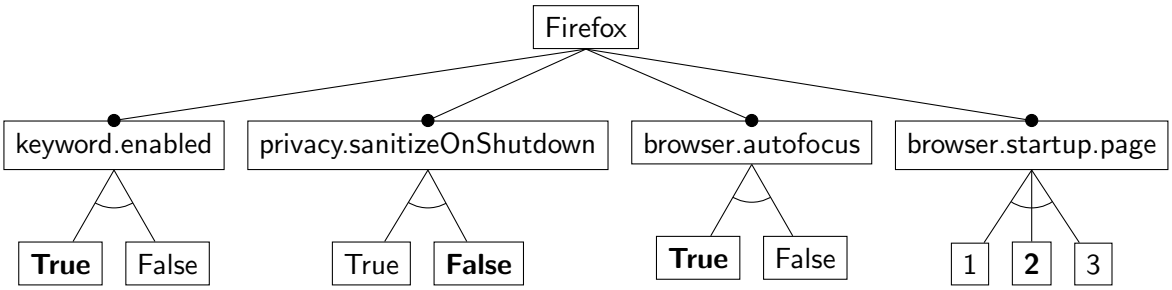

Figure 4.5: Minimized Configuration 2

The failure encountered is caused by *browser.startup.page* being set to three, same as before. This means that the first configuration in figure 4.2 still fails, but the second one in figure 4.3 will pass. Because of this random configuration two serves as a workaround. Looking at random configuration two we can see that it has three reconfigurations from the starting configuration. We want to minimize this to find the minimal reconfigurations necessary to avoid the failure.

The first thing the minimizer does is rerun the configuration to make sure that it passes. Then it splits the reconfigurations into two groups, and fills in the blanks with the starting configuration values. This will result in two minimized configurations. One in figure 4.4 and the other in figure 4.5

The first minimized workaround will fail and the second one will pass. This means that *keyword.enabled set* to false can be discarded as a reconfiguration as it is not necessary to create a workaround. The process is repeated with the remaining two

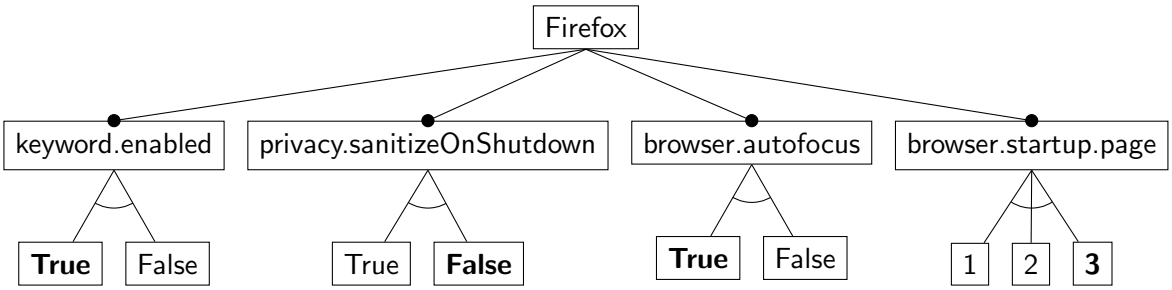

Figure 4.6: Minimized Configuration 3

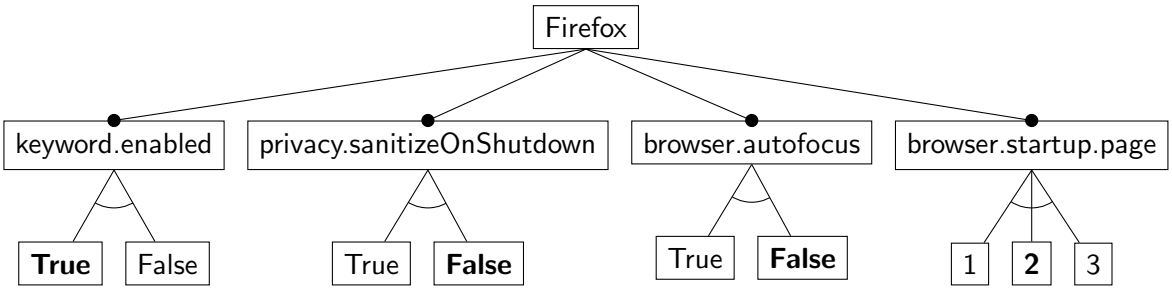

Figure 4.7: Minimized Configuration 4

reconfigurations and two more minimized configurations are generated. One in figure 4.6 and the other in figure 4.7.

Minimized configuration three will fail its test and minimized configuration four will pass. This means that *browser.autofocus* set to true can be discarded as it is not necessary for a workaround. This leaves only *browser.startup.page* set to two, which as we know will avoid the failure that occurs when *browser.startup.page* is set to three. The final result is the workaround configuration shown in figure 4.8.

The minimizer ran five tests once the failure was encountered. One to replicate the test, and then four during the course of minimization. Compared to one-hop the random plus minimizer actually performs worse in this case. However, consider that the size of this feature model is quite small, only four features with 9 configuration options. When dealing with a feature model with hundreds of options the one-hop algorithm will scale linearly. While searching for a one-hop reconfiguration workaround

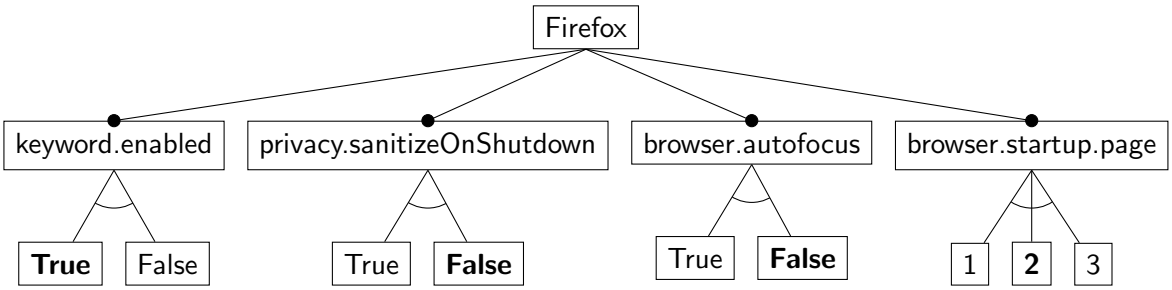

Figure 4.8: Workaround Configuration

random plus the minimizer will take logarithmic time with respect to the size of the feature model. Finding larger then one-hop workarounds will take longer, but this is countered by the fact that the one-hop algorithm cannot find any workarounds larger then one-hop.

4.2.4 Multi-Minimizer

One of the problems with the randomization algorithm followed by the minimizer algorithm is that it might contain multiple workarounds. To find additional workarounds we implemented a multi-minimizer that finds these additional workarounds. For example if the partition c_1 is a workaround then using the minimizer algorithm C is reduced to c_1 and the process restarts. Now, not only is C reduced to c_1, but once C has been minimized c_2 is also checked to see if it is a workaround. In the original algorithm it didn't matter if there are multiple ways to recreate the failure, only finding one was enough. However part of the Failure avoidance algorithm is the guard that can prevent further failures. Improving the guard means finding as many workarounds as possible and from the original research we know that there are bugs that have multiple one-hop workarounds. Using the multi minimizer means more of these workarounds can be discovered and minimized.

The major problem with the multi minimizer is the increased cost. Obviously

checking each of the partitions instead of just the first passing one will mean that the algorithm takes longer. However if no workaround is found then obviously the algorithm is not called in the first place. This means that the multi minimizer will only add additional workarounds to bugs that already have at least one workaround, and the additional cost will only be added when dealing with bugs that have multiple workarounds.

4.2.5 Covering Array

Assuming that there are workarounds beyond two-hop, we want to be able to explore some systematic ways to find these, since random may or may not find these workarounds. And we want to do this efficiently. For this we turned to *covering arrays*.

Covering arrays have been used for testing highly configurable software systems [32,44,55]. The idea is that it is impossible to test every combination of configurations, however you could test every x pair combination of configurations, where x is some number greater then one. The idea is that testing the combinations will be sufficient to find the hidden bugs that manifest only with select configurations.

What we want to do is create a covering array for the feature model, and then each time a failure is encountered check the covering array for a workaround. If a workaround is detected then the configuration will be reduced using the minimizer or the multi-minimizer. There has been work done on combining delta debugging with covering arrays [35] and using covering arrays to find a minimal failing test [40], though our work will focus on using covering arrays to more systematically explore the configuration space and then using delta debugging to minimize the found configuration workarounds.

Initially it was decided to try two-way and three way covering arrays. The two-way covering arrays will take about 15 tests, and the three way around 60 to cover the GCC feature model. This means that the time costs for the covering arrays should be lower then the expensive random tests. In addition because the covering arrays are systematically built the number of duplicate workarounds should be reduced when compared to the random tests.

4.3 Case Study

We conducted a case study to determine if the new algorithms, improve on the original one or two-hop algorithms for failure avoidance. We first ask if there are workarounds beyond what can be found only with one or two-hop, as well as evaluate the cost of running these algorithms on our data. In this study we answer three research questions.

RQ1. Do we find failures that can be avoided only beyond the one or two-hop distance?

RQ2. Can we find additional (new) workarounds for failures beyond those found by the one or two-hop algorithm.

RQ3. What are the tradeoffs in terms of effectiveness and efficiency between the different failure avoidance algorithms.

4.3.1 Study Design

Objects of Study. Our evaluation of the new failure avoidance algorithms was done on several versions of GCC [2]. We chose GCC because it is a highly-configurable system with a large and active user base. The other advantage of using GCC was that the evaluation for the one and two-hop algorithms had already been done. As

long as we can reasonably replicate the results, we have a nice baseline from which to compare our new algorithms. The three versions of GCC used were 4.4.0, 4.4.1 and 4.4.2. All of these versions were released in 2009, and they are all quite large, each exceeding 23 million lines of code.

We were able to use the feature model and test suite from the original experiments. The feature model was kept restricted to command line arguments and contained a small number of configuration options which represent adjusting the optimization level for GCC. All three versions share a feature model. The test suite was generated by collecting failures from GCC's public bug database. It started with 360 failures and in the original work was pruned to 237. The failures removed were either platform dependent, nondeterministic or no indisputable oracle could be used.

Preliminary Study. The first part of the study is to replicate the results from the original work. The three versions of GCC used, 4.4.0, 4.4.1 and 4.4.2, were downloaded and installed on sandhills. The tests were then run and the results compared. Some variation was expected as the underlying architecture would be different. For GCC 4.4.0 127 bugs could be replicated, and workarounds were found for 29 of those bugs using the one-hop algorithm. The original work had 137 bugs replicable, with 31 having workarounds.

To understand this difference, the feature model of GCC was examined and a few discoveries were made. There were some reconfiguration options, such as $-f-section-anchors$, that upon closer inspection were dependent on the architecture of the system. This meant that a workaround that included $-f-section-anchors$ might not be a workaround on a different architecture. These reconfiguration options were locked for future experiments, though there is some interesting work possible which explores the idea of architecture dependent workarounds.

Another thing that was looked at was constraints. When the constraints of the

GCC system were included in the feature model the number of workarounds found for the one-hop algorithm were drastically reduced. However experiments run using the two-hop algorithm managed to perform close to the unconstrained results. Closer inspection into GCC showed that the system is smart enough to include needed configurations when they are not specifically declared. This means that if a one-hop workaround uses a reconfiguration that requires an optimization level to be set, the optimization level will be adjusted automatically if it is not included. Essentially the workaround requires two reconfigurations, but one will suffice as the system will automatically include the other. Because the unconstrained results more closely resemble original GCC results, it was decided to move forward using an unconstrained feature model for future tests.

The final results for GCC 4.4.0 are 127 bugs replicated, with 27 bugs with workarounds for both the one-hop and the two-hop algorithm. This is close enough to the original results that it will serve as a baseline for comparing to new algorithms for finding workarounds. For GCC 4.4.1 there were 123 bugs replicated with 26 bugs with workarounds for both the one-hop and two-hop algorithm. Finally in GCC 4.4.2 there were 110 bugs replicated with 22 bugs with workarounds for both the one-hop and two-hop algorithms.

4.3.2 Independent and Dependent Variables

The independent variables in our study are the variants of the failure avoidance algorithm. First we use the one and two-hop algorithms. We also use the random algorithm with 10, 25, 50, 100 an 150 iterations. For each we applied both the minimizer and multi-minimizer. We also built two and three-way covering arrays for our application using the CASA tool [26] Our dependent variables are the number of

workarounds and the time to find the workarounds.

4.3.3 Results

The results of the one and two-hop algorithms, the random algorithm trials, and the covering array trials are shown in table 4.1. The table displays the average number of workarounds found for each bug that had a workaround for each of the algorithms tried. The one and two-hop were only run once as the results will not change. The other algorithms rely on some degree of randomness so they were run for five trials each and the results displayed are an average of those trials.

For the random algorithm, 10 iterations for both the minimizer and multi-minimizer averaged 26.2 and 26.4 bugs with workarounds respectively. This means that random at this level is not finding as many bugs with workarounds as the one or two-hop algorithms. In addition the time cost for running random with the minimizer came out to a little over an hour, and the multi-minimizer took almost an hour and a half. More than the cost of one-hop at just over half an hour.

Running random with 25 iterations increases the number of bugs with workarounds found past 27, however it is still not complete compared to the one-hop algorithm. In fact increasing the iterations to 150 gives an average number of bugs with workarounds of 29.8 for the minimizer and 29.6 for the multi-minimizer and still is not guaranteed to find all of the one-hop workarounds we know exist. At most 30 bugs with workarounds are found, and the new workarounds are three-hops away from the starting configuration.

Finding these workarounds is very interesting because it provides evidence that there are workarounds that could not be discovered using either the one or two-hop approach. The two-hop algorithm took over 139 hours which means that a three-

hop version of the algorithm is would take at least 900 days running on one machine. Parallelization could be used to decrease this computation time, however it could also be used on any of the other approaches. Considering the 150 iteration random with multi-minimizer took a little over 20 hours and will scale to larger feature models it makes sense that for finding these three-hop workarounds a brute force approach is not the most effective..

Because the random algorithm could not find all of the one-hop workarounds even with 150 iterations, a two-way covering away was used. The idea is that each two-way pair of reconfiguration options should guarantee that all the one-hop workarounds were found. When the tests were run though, using five different randomly generated two-way covering arrays an average of 23 bugs with workarounds were found. These results are not as good as the one or two-hop algorithms. We were not expecting the covering array to miss one-hop workarounds as it is guaranteed to have all of the two-way combinations. Looking into it we discovered that there is configuration masking going on. A situation exists where some reconfigurations will affect others, such as the $-fschedule-insns$ and $-fschedule-insns2$ example from above, and even though all two-way combinations are present they might not be tested.

To overcome this problem we first tried combining two sets of two-way covering arrays (2Dway). There are many different ways to reach all two-way combinations, so combining two sets could help reduce the masking effect. The results were better. Costing only about three hours on average the double stacked two-way covering arrays were comparable to the random 100 iteration multi-minimized tests at a fraction of the time cost. The results of the three way covering array were similar to that as well. The time cost and the number of workarounds found were very close, even though the three way covering array has 30 additional tests to run.

The time to first workaround was also examined. This represents the case where

instead of finding all workarounds for a bug we only care about the first one. The quicker a test can return that a workaround was found the better. Random performed well at almost 17 minutes, which is less then half the time it took for the one-hop algorithm to run. However, the covering arrays performed even better with the two-way at just over 15 minutes, and the three way down to about 12 minutes 30 seconds. This result shows another area where the covering array seems to outperform the random algorithm.

4.3.4 Number of Workarounds Found

There are 29 failures that have at least one workaround, and of these 27 could be found using the one-hop algorithm. In addition there are some failures where it seems as though there is a limit on the number of workarounds to be found, such as bug 25689, 25733 and 41016. Even with a large number of tests run the number of found workarounds always seems to be the same number.

As expected, the more tests run the higher the chance of finding additional workarounds. Looking at the average, as the number of iterations increase the number of workarounds found increases as well. It also shows that the multi-minimizer finds more workarounds on average then just the minimizer. This makes sense as we know there are a number of one-hop workarounds and the multi minimizer is capable of finding all of them that are present in one random configuration. In addition it appears that on average the random multi minimizer for 100 and 150 iterations are equal. This is interesting as an increase in 50 random configurations does not necessarily yield additional workarounds.

If we examine bug 39794 using the three way covering array, we see that 18 workarounds can be found, while the random algorithms find only 17 workarounds at

	Brute Force		Covering Array			Random minimizer						Random multi minimizer					
BugNum	BF/1Hop	BF/2Hop	CA/2Way/	CA/2DWay/	CA/3Way/	gccRM/10FI/	gccRM/10I/	gccRM/25I/	gccRM/50I/	gccRM/100I/	gccRM/150I/	gccRMM/10I/	gccRMM/25I/	gccRMM/50I/	gccRMM/100I/	gccRMM/150I/	
25689	2	2	2	2	2	1	2	2	2	2	2	2	2	2	2	2	
25733	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
31029	2	2	0	1.6	0.8	0.2	0	0.2	1.2	1	1.8	0.8	0.6	1	1.4	2	
36902	3	3	0	0.8	1.4	0.2	0.2	0.4	0.6	0.6	1.6	0	0.4	0.2	1.6	1	
38540	1	1	2.4	3.6	5	1	1.6	3.6	3.8	4.8	4.8	3.4	4.2	4.8	5	5	
38541	1	1	2.4	3.6	5	1	2.4	3.4	3.8	4.6	5	2.6	3.8	4.8	5	5	
38808	7	12	3.4	11.8	12.8	1	2.6	4.4	5.6	6.8	7.8	4.2	5.8	8.2	9.4	10.6	
39794	14	16	12.6	17.4	18	1	4.6	7.4	10	10.4	11	14.8	16.4	17.2	17.8	17.8	
40321	6	11	7.2	13.4	13	1	5	5.6	6	6.8	7.6	7.6	9.4	11	11.8	11.8	
40389	2	2	1	2	2	1	2	2	2	2	2	1.8	2	2	2	2	
40579	11	11	9.6	11.8	12	1	4.8	7.6	6.8	8.8	9.2	10.6	11.6	11.6	12	12	
40796	2	5	3	10.6	11.4	1	2.2	4	5.6	8.4	8.2	3.8	7	8.6	11.6	10.8	
41016	1	1	0	1	1	1	1	1	1	1	1	0.8	1	1	1	1	
41153	2	2	1	2	2	1	2	2	2	2	2	2	2	2	2	2	
41183	5	5	4	5	5	1	3.6	5	5	5	5	5	5	5	5	5	
41403	1	3	0.4	2.8	4	1	1.2	3	2.6	3.4	4.8	2.2	2	3.2	4	4	
41623	4	4	1.8	4	4	1	2.4	3.8	4	4	4	2.8	3.6	4	4	4	
41630	3	3	1.4	3.6	4	1	2.8	3.4	3.4	4	4	3.2	4	4	4	4	
41643	4	6	5	8.8	9	1	4.2	4.2	6.2	6.2	6.4	6.4	7.8	8.8	9	9	
41646	3	4	2.8	8.2	8.4	1	3.2	5	5.6	6.6	7.4	4.2	6.8	8.4	8.8	8.8	
41917	1	1	0	1.6	2	0.8	1.6	1.6	1.6	2	2	0.8	1.4	1.8	2	2	
42049	7	7	5	7	7	1	4.8	6	6.4	6.8	7	7	7	7	7	7	
42231	7	8	5.8	8	8	1	4.6	6.2	7.4	7.8	8	7.6	8	8	8	8	
42488	5	5	0	2.4	2.8	0.4	0	0.8	0.8	2.2	3.4	0	1.4	2.4	4.2	3.8	
42542	4	4	3.2	8	8.8	1	3.8	5	5.4	6.2	6.2	4.8	7	8	9	9	
42614	4	6	3.8	7	7	1	4	5.8	6.6	6.8	6.8	5.8	7	7	7	7	
42667	3	3	0.8	3	3	1	3	3	3	3	3	2.8	3	3	3	3	
43024	5	5	3	5	5	1	4.4	4.8	5	5	5	4.6	5	5	5	5	
40749	0	0	1.4	2.6	4	0.8	2	2.4	4	4	4	2.4	2.8	3.8	4	4	
8045	0	0	0	0.8	1.6	0.4	0	0.8	1.8	2.2	2	0.8	0.6	1.6	2.6	3.6	
Average	3.70	4.47	2.80	5.35	5.70	0.89	2.57	3.51	4.01	4.51	4.80	3.86	4.65	5.21	5.67	5.71	

Table 4.1: Number of Workarounds Found for Each Bug by Technique in GCC 4.4.0

most. This implies that there are more workarounds to be found, but the necessary configuration to see the workaround isn't being generated by the random algorithm. Looking into bug 39794's workarounds deeper shows that the workaround that the three way covering array finds but random misses is a four-hop reconfiguration that uses the options $-fwrapv$, $-fno-tree-vrp$, $-fno-tree-ccp$, and $-fno-tree-dominator-opts$. For a random configuration to have all of these options statistically is a little over 6%. This isn't even considering the masking of configuration options that we know exists in gcc. The masking occurs when the selection of one configuration can alter or change another configuration. This can reduce the odds of finding the four-hop workaround even further, which means for these complicated multi-hop workarounds, random will take a large number of iterations and may never find them.

4.3.5 Number of Hops for Workarounds Found

Table 4.2 shows the number of hops for each workaround found over all algorithms testing GCC 4.4.0. Bug 40749 and 8045 only have workarounds that are three-hops from the starting configuration. The original failure avoidance algorithm would have needed to exhaustively search all three-hop combinations to find these new workarounds. This would require 159 *158 * 157 tests, which comes out to a total of at most 3944154 test cases. This shows how an exhaustive technique is limited with even a small increase in the number of hops to be checked. The workarounds in question involve changing the optimization level of GCC, and then turning on two related configuration options.

Certain configuration options are related to each other, such as affecting similar aspects of the program. For instance a workaround might require $-fbranch-target-load-optimize$ and $-fbranch-target-load-optimize2$. $-fbranch-target-load-optimize2$ requires

BugNum	1Hop	2Hop	3Hop	4Hop	5Hop
25689	2	0	0	0	0
25733	1	0	0	0	0
31029	2	0	0	0	0
36902	3	0	0	0	0
38540	1	0	4	0	0
38541	1	0	4	0	0
38808	7	5	0	1	0
39794	14	2	1	1	0
40321	6	14	5	0	0
40389	2	0	0	0	0
40579	11	0	0	0	1
40796	2	3	8	0	0
41016	1	0	0	0	0
41153	2	0	0	0	0
41183	5	0	0	0	0
41403	1	2	0	1	0
41623	4	0	0	0	0
41630	3	0	1	0	0
41643	4	2	3	0	0
41646	3	1	2	2	1
41917	1	0	0	0	1
42049	7	0	0	0	0
42231	7	1	0	0	0
42488	5	0	0	0	0
42542	4	0	1	3	1
42614	4	2	1	0	0
42667	3	0	0	0	0
43024	5	0	0	0	0
40749	0	0	4	0	0
8045	0	0	12	0	0
Average	3.70	1.07	1.53	0.27	0.13

Table 4.2: Number of Hops for all Workarounds Found for Each Bug in GCC 4.4.0

$-fbranch-target-load-optimize$ and an optimization level higher then zero to be selected. This workaround could not be found without checking all three of these options.

We also see a handful of bugs with workarounds that require three, four or even five-hops. The greatest number of hops found for a workaround is five. Looking at GCC 4.4.0 this occurs for four different bugs, 40579, 41646, 41917 and 42542. two of the bugs have workarounds for every level of hop between one and five, and the other

two only have one-hops and a five-hop workaround. Lets take a closer look at some of these bugs.

The one-hop workarounds for 40579 are *−fpeel−loops* , *−funroll−loops* , *<bundle−only feature 0 of [null, null, null, null, null]>* , *<bundle−only feature 1 of [null, null, null, null, null]>* , *−fwrapv, <bundle−only feature 3 of [null, null, null, null, null]>* , *−ftrapv* , *−fno−tree−vrp* , *−fno−tree−loop−optimize* , *−fno−ivopts* and *−funroll−all−loops*. From this we can see that there are lots of configuration options available to create a workaround, from adjusting the optimization level, messing with loops, changing trees, etc. Looking at these options it would seem that there are many ways to avoid this particular bug. The five-hop workaround is *−fno−tree−dce*, *−fno−tree−scev−cprop*, *−fno−tree−ch*, *−fno−tree−copy−prop* and *−fno−tree−dominator−opts*. Every single one of these configuration options deals with trees in some way, but none are the the same option from the one-hop workarounds. This makes sense because if any of the options in the five-hop workaround could be found in the list of one-hop workarounds we would expect the configuration to be minimized.

The single one-hop workaround for bug 41917 is *<bundle−only feature 0 of [null, null, null, null, null]>*. This translates to setting the optimization level to zero. The five-hop workaround is *−fno−forward−propagate*, *−fno−tree−vrp*, *−fno−gcse*, *−fno−tree−ccp* and *−fno−tree−dominator−opts*. Notice how each of these configuration options involves turning off some flag. Each is a *−fno* option. Essentially, changing the configuration level to zero will avoid the failure because it turns off these five options. This five-hop workaround shows us exactly what flags are causing the problem, and gives a developer working on fixing the failure a much clearer picture of where the failure is. If you consider the fact that turning the optimization level to zero affects many configuration options n then the workaround with more hops is a subset of the workaround with less hops in this case.

	Brute Force		Covering Array			Random minimizer					Random multi-minimizer				
Time	1Hop	2Hop	2Way	2DWay	3Way	10I	25I	50I	100I	150I	10I	25I	50I	100I	150I
Hours	0.60	139.27	0.77	2.71	3.11	1.18	2.20	4.58	9.08	14.18	1.45	3.27	6.77	13.42	20.93
Minutes	36.00	8356.00	46.20	162.80	186.80	71.00	132.00	275.00	545.00	851.00	87.00	196.00	406.00	805.00	1256.00
TestCases	159	25122	15	30	60	10	25	50	100	150	10	25	50	100	150

Table 4.3: Time for Each Technique to Run

Time	CA2Way	CA3Way	Random
Hours	0.25	0.21	0.28
Minutes	15.20	12.40	16.80

Table 4.4: Time to First Workaround

4.3.6 Time for Techniques to Run

The times were measured for each technique to run and displayed in Table 4.3. In addition we look at the number of test cases that were run. For the case of the covering array a test case represents a possible workaround configuration, and then additional tests would be run by the minimizer to find the minimal workaround configuration.

The longest test we ran was clearly the two-hop algorithm, taking almost 7 times longer then the longest random technique. This shows just how infeasible a brute force technique would be for finding those three-hop workarounds for bugs that have no one or two-hop workarounds.

The covering array performed fairly well, with the two-way only taking ten minutes longer on average then the one-hop technique. Looking at our results from earlier, we know that the two-way covering array is not very good at finding all of the workarounds discovered by the one-hop. The double stacked two-way covering array took almost four times longer with only an additional 15 tests. This increase in time was likely caused by a large number of duplicate configurations that required the minimizer to be used more often. The three way covering array performs only a little slower, taking an additional 20 minutes, however it adds 30 test cases. This shows that even though there is an increase in test cases, they are spread out around

the configuration space, and there are fewer minimizations necessary overall.

Random is compared with the minimizer and the multi-minimizer. Obviously the more iterations the longer the tests will take to run. The multi-minimizer also takes longer than the minimizer due to an increase in the number of times a configuration is checked to see if it is a workaround. The minimizer will not know if it is working on minimizing a duplicate workaround until the end, which means that a large amount of time can be wasted during the minimization step. From the earlier table we know that there is not much difference between the 100I MM and 150I MM tests, however the additional 50 tests mean a seven hour increase in run time. Even with the smallest run of ten iterations and the regular minimizer the random tests took twice as long as the one-hop algorithm.

We also looked at time to first workaround for the covering array and random algorithms as shown in Table 4.4. The three way covering array is actually the fastest, with an average time of just under 12 and a half minutes. The results from random aren't bad either, with a response on average in the first 17 minutes. If all a user was concerned about was finding a workaround, any workaround, as quickly as possible then the covering array or random algorithm work much faster than even the one-hop algorithm, taking only half the time. In addition the time to first workaround will be shorter for a failure with multiple workarounds as the odds of finding any workaround increases. An experiment to calculate the average time to first workaround using the one-hop algorithm didn't make sense, as the order of the feature model will determine when the workarounds are discovered.

	Brute Force		Covering Array			Random minimizer					Random multi-minimizer				
Workarounds found in	1Hop	2Hop	2Way	2DWay	3Way	10I	25I	50I	100I	150I	10I	25I	50I	100I	150I
all trials	28	28	25	28	29	26	26	28	29	30	25	27	29	30	30
at least one trial	28	28	30	30	30	27	30	30	30	30	27	30	30	30	30

Table 4.5: Number of Bugs with at Least One Workaround Found

	Brute Force		Covering Array			Random minimizer						Random multi minimizer				
BugNum	1Hop	2Hop	2Way	2DWay	3Way	10FI	10I	25I	50I	100I	150I	10I	25I	50I	100I	150I
25689	2	2	2	2	2	1	2	2	2	2	2	2	2	2	2	2
25733	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
31029	2	2	0	1.6	0.8	0.2	0.4	0.6	1	1.2	1.6	0	0.8	0.8	1.4	2
36902	3	3	0	0.8	1.4	0	0	0	0.4	0.6	0.8	0	0.2	0.2	1.8	1.8
38540	1	1	2.4	3.6	5	1	2	3	4	4.8	5	3.2	4.2	4.6	5	5
38541	1	1	2.4	3.6	5	1	1.8	2.4	3.2	5	5	2.4	4.2	4.8	5	5
38808	7	12	3.4	12.4	12.4	1	3	3.6	5.4	6.6	7.6	4	7.6	8.2	9.4	11.4
40321	6	11	7.2	13.2	13	1	3.6	5.4	6.2	7	7.6	8	10.4	10.8	12	12.8
40796	2	5	2.6	5	5	1	2.8	3.8	4.4	5	5	3.8	4.6	5	5	5
40924	9	14	8.6	13.2	15.4	1	5	7	8.2	9.2	10.2	10.6	13.2	14.4	16.2	16.2
41016	1	1	0	1	1	1	1	1	1	1	1	0.8	1	1	1	1
41153	2	2	1	2	2	1	2	2	2	2	2	1.8	2	2	2	2
41183	5	5	4	5	5	1	2.8	4.6	5	5	5	5	5	5	5	5
41403	1	2	0.4	2.4	3	0.8	2	1.6	2.6	3.2	3.4	1.2	2.6	3	3	3
41619	8	8	6.6	8	8	1	4.2	5.8	7	8	7.8	7.4	8	8	8	8
41630	3	3	1.4	3.6	4	1	2	3.4	3.4	4	4	3.4	4	3.8	4	4
41643	4	6	5	8.8	9	1	4.2	5.2	5.2	5.8	6.4	6.6	8	9	9	9
41646	3	4	3.6	8.8	9	1	4.2	4.6	5.6	6.6	6.8	5.6	7.6	8.8	9	9
41917	1	1	0	1.6	2	1	1.4	1.6	1.8	1.6	2	1.4	1.8	1.4	2	2
42049	7	7	5	7	7	1	4.6	5.8	6.8	6.8	7	6.8	6.8	7	7	7
42231	7	8	5.8	8	8	1	5.2	6	7.6	7.8	8	6.8	8	8	8	8
42488	5	5	0	2.4	2.8	0	0.4	0.4	1	2	3	1	1.8	3.2	4	4.6
42542	4	4	3.2	8	8.8	1	3.4	5	5.8	6.6	6.6	5.4	7.4	8.6	8.8	9
42614	4	6	3.8	7	7	1	5	5.4	6.2	7	7	6.2	6.8	7	7	7
42667	3	3	0.8	3	3	1	2.6	3	3	3	3	2.8	3	3	3	3
43024	5	5	3	5	5	1	3.4	4.8	5	5	5	4.8	5	5	5	5
40749	0	0	1.4	2.6	4	0.8	1	2.8	3.6	4	4	2.2	3.2	3.4	4	4
8045	0	0	0	0.8	1.6	0	0	0.6	0.4	1.2	2.8	1	1	2.4	4.8	4.4
Average	3.46	4.36	2.66	5.05	5.40	0.85	2.54	3.30	3.89	4.39	4.66	3.76	4.69	5.05	5.48	5.61

Table 4.6: Number of Workarounds Found for Each Bug by Technique in GCC 4.4.1

BugNum	1Hop	2Hop	3Hop	4Hop	5Hop
25689	2	0	0	0	0
25733	1	0	0	0	0
31029	2	0	0	0	0
36902	3	0	0	0	0
38540	1	0	4	0	0
38541	1	0	4	0	0
38808	7	5	0	1	0
40321	6	12	2	2	0
40796	2	3	0	0	0
40924	9	5	2	1	2
41016	1	0	0	0	0
41153	2	0	0	0	0
41183	5	0	0	0	0
41403	1	1	0	1	0
41619	8	0	0	0	0
41630	3	0	1	0	0
41643	4	2	3	0	0
41646	3	1	3	2	0
41917	1	0	0	0	1
42049	7	0	0	0	0
42231	7	1	0	0	0
42488	5	0	0	0	0
42542	4	0	1	3	1
42614	4	2	1	0	0
42667	3	0	0	0	0
43024	5	0	0	0	0
40749	0	0	4	0	0
8045	0	0	12	0	0
Average	3.46	1.14	1.32	0.36	0.14

Table 4.7: Number of Hops for all Workarounds Found for Each Bug in GCC 4.4.1

4.4 Discussion of Results

The first question attempted to answer in this case study, is whether or not there exists any additional workarounds beyond one or two-hops. In addition we asked if there are any failures that do not have a one or two-hop workaround, but do have a workaround greater then two-hops away from the starting configuration. By using a random algorithm with a minimizer we have seen that there are three-hop workarounds for bugs that have no one or two-hop workaround. We also discovered workarounds up to five-hops away from the starting configuration.

	Brute Force		Covering Array			Random minimizer						Random multi minimizer				
BugNum	1Hop	2Hop	2Way	2DWay	3Way	10FI	10I	25I	50I	100I	150I	10I	25I	50I	100I	150I
25689	2	2	2	2	2	1	1.8	2	2	2	2	2	2	2	2	2
25733	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
31029	2	2	0	1.6	0.8	0.2	0.4	0.2	0.6	1.6	1.8	0	0.8	1.4	1.8	2
36902	3	3	0	0.8	1.4	0	0.2	0.4	0.8	0.2	0.6	0	0.4	0.2	0.8	1.2
38540	1	1	2.4	3.6	5	1	1.2	2.6	3.8	4.6	4.6	2	3.8	4.8	5	5
38541	1	1	2.4	3.6	5	1	2.6	2.4	4.4	4.6	4.8	2.6	3.8	4.4	4.8	5
38808	7	12	3.4	12.2	12.4	1	2.8	3.2	5	6.6	7.4	3.8	6.2	7.8	9.6	10.6
40796	2	5	2.6	5	5	1	2.2	3.4	5	4.8	5	3.6	4	5	5	5
41153	2	2	1	2	2	1	2	2	2	2	2	2	2	2	2	2
41183	5	5	4	5	5	1	3.6	5	5	5	5	4.8	5	5	5	5
41403	1	2	0.4	2.4	3	0.8	1.4	2.8	2.8	3.8	3.6	1.4	1.8	3	3	3
41619	8	8	6.6	8	8	1	5.4	6.4	6.8	7.6	8	7.8	8	8	8	8
41630	3	3	1.4	3.6	4	1	3.2	3.2	3.6	4	3.8	3.8	4	4	4	4
41643	4	6	5	8.8	9	1	4.2	4.4	5.4	7	7.6	6.8	7.8	8.8	9	9
41917	1	1	0	1.6	2	0.8	0.8	1.2	1.8	1.6	2	1	1.6	1.6	2	1.8
42049	7	7	5	7	7	1	4.2	5.8	6.8	6.6	7	6.4	7	7	7	7
42231	7	8	5.8	8	8	1	5.2	6.4	7.4	7.6	8	7.8	7.8	8	8	8
42488	5	5	0	2.4	2.8	0.2	0.2	1	2.2	2.4	2.8	0.2	2.6	4	3	4.2
42542	4	4	3.2	8	8.8	1	4.2	5.2	5.2	6.2	6.6	5.2	7.6	7.8	8.6	9
42614	4	6	3.8	7	7	1	4.6	5.8	6.6	6.6	6.8	5.8	7	7	7	7
42667	3	3	0.8	3	3	1	2.8	3	3	3	3	3	3	3	3	3
43024	5	5	3	5	5	1	4	4.8	5	5	5	4.8	5	5	5	5
8045	0	0	0	0.8	1.6	0	0.6	0.4	0.2	2.2	2.6	0	1.2	1.4	5	5
Average	3.39	4.00	2.34	4.45	4.73	0.83	2.55	3.16	3.76	4.17	4.39	3.30	4.06	4.44	4.77	4.90

Table 4.8: Number of Workarounds Found for Each Bug by Technique in GCC 4.4.2

The second question dealt with comparing new techniques to the original brute force. We wanted to find new workarounds, find them quicker, and guarantee that all of the workarounds found using brute force can be discovered as well. Finding new workarounds is easy. Finding them quicker was much harder to do, as the act of minimizing any workaround found takes a great deal of time. Compared to the brute force where each workaround found will already be minimal. The other issue is guaranteeing to find all of the workarounds that the brute force can detect. The problem with this is some of the workarounds are more difficult to detect due to masking of configuration options. This means that unless an exhaustive approach is used, or a great deal of time is spent, it might not be possible to guarantee all workarounds from the one or two-hop algorithm are found.

BugNum	1Hop	2Hop	3Hop	4Hop	5Hop
25689	2	0	0	0	0
25733	1	0	0	0	0
31029	2	0	0	0	0
36902	3	0	0	0	0
38540	1	0	4	0	0
38541	1	0	4	0	0
38808	7	5	0	1	0
40796	2	3	0	0	0
41153	2	0	0	0	0
41183	5	0	0	0	0
41403	1	1	0	1	0
41619	8	0	0	0	0
41630	3	0	1	0	0
41643	4	2	3	0	0
41917	1	0	0	0	1
42049	7	0	0	0	0
42231	7	1	0	0	0
42488	5	0	0	0	0
42542	4	0	1	3	1
42614	4	2	1	0	0
42667	3	0	0	0	0
43024	5	0	0	0	0
8045	0	0	12	0	0
Average	3.39	0.61	1.13	0.22	0.09

Table 4.9: Number of Hops for all Workarounds Found for Each Bug in GCC 4.4.2

4.4.1 Summary

In this section we evaluated several algorithms for finding workarounds. The random algorithm did find new workarounds, but it was not guaranteed to locate all of the one-hop workarounds found through a bounded exhaustive search. Even though the covering arrays could not guarantee finding all of the one-hop workarounds, they found a large number of two-hop and beyond workarounds with five times the time cost. Compare this to the fact that the two-hop algorithm takes 236 times the time cost of running the one-hop algorithm, significant strides have been made.

For research question one we asked if there exists failures avoided only by three-hop or greater workarounds. For GCC version 4.4.0 looking at Table 4.1 the random

and covering array versions of the failure avoidance algorithm were able to find a three-hop workaround for bug 40749 and bug 8045. These bugs do not have a one or two-hop workaround and will not be found by the one-hop or two-hop algorithm. Version 4.4.1 of GCC in table 4.6 has the same workarounds found and version 4.4.2 in table 4.8 has three-hop workarounds found for bug 8045 using our new techniques. We can answer question one in the affirmative because we discovered workarounds, in all three versions of GCC tested, for bugs that do not have a one or two-hop workarounds.

For research question two we asked what additional workarounds can be discovered by exploring beyond a two-hop space. For GCC version 4.4.0 looking at table 4.2 we can see that the average number of one-hop workarounds for a bug is 3.7. This fits in with what we know about there being multiple ways to workaround a failure. The average number of two-hop workarounds drops down to just over one per bug. A large part of this is due to the 14 two-hop workarounds for bug 40321. For the three-hop workarounds the number jumps up, to 1.53 workarounds per bug. The fact that there are more three-hop workarounds for bugs then two-hops is interesting, because it means that an algorithm that can explore beyond the two-hop space has a larger pool of potential workarounds to discover then previously thought.

There are an average of 0.27 four-hop workarounds and 0.13 five-hop workarounds per bug as well. The fact that workarounds exists at such a distance from the starting configuration is quite interesting, as it means that a failure is tied to at least that number of features. However, because all of the bugs with four-hop and five-hop workarounds have at least one one-hop workaround as well, there is a much easier way to avoid the failure. Consider the case of bug 41917. The one-hop workaround involves shutting off all optimization while the five-hop workaround involves shutting off five specific optimizations. If the user wanted to avoid the bug, but not remove

all optimizations, then the five-hop workaround would actually be preferred.

For GCC version 4.4.1 in Table 4.7 the numbers are similar to version 4.4.0. One-hop the most number of workarounds per bug at 3.46, a drop to 1.14 for two-hop, an increase to 1.32 for three-hop and finally a drop to 0.36 and 0.14 for four and five-hops respectively. GCC version 4.4.2 in table 4.9 has the number of workarounds per bug at 3.39 for one-hops, 0.61 for two-hops, 1.13 for three-hops, 0.22 for four-hops and 0.09 for five-hops. This means that the pattern of more three-hop workarounds per bug then two-hop workarounds holds for all versions of GCC tested and each version tested has four and five-hop workarounds found. For question two we can answer that there are more three-hop workarounds found then two-hop workarounds, and four and five-hop workarounds exist which can in cases be more useful to the user then the comparable one-hop workaround.

For research question three we asked what is the most effective and efficient algorithm for detecting workarounds. What we were looking for is a balance between an algorithm that runs quickly (efficient) and finds at least one workaround for every bug that has one (effective). Looking at GCC 4.4.0 In table 4.3 the results for the time to run each technique are listed. The one-hop algorithm runs in only 36 minutes and is guaranteed to find all one-hop workarounds. However because it cannot find any workarounds beyond one-hop and we know there are bugs with three-hop workarounds as seen in table 4.1 only it loses some effectiveness. Table 4.5 shows for each technique if at least one workaround for a bug with a workaround was found and if it was found during all five test runs or on at least one of the tests. The one-hop algorithm only finds workarounds for 28 of the 30 bugs with workarounds. The two-hop algorithm takes very long to run and at almost 140 hours its efficiency is far too low to make it a viable option. It also only finds workarounds for 28 of the 30 bugs with workarounds

The random algorithm with the minimizer takes less time on average than random with the multi minimizer and the same number of iterations. Because of this the random with minimizer is more efficient then random with the multi minimizer. The multi minimizer will allow a single iteration to return multiple workarounds. This essentially means that if the iteration does not contain any workarounds, then none will be found, regardless of the minimizer used. Because we are concerned with finding any workaround for a bug, the multi minimizer isn't any more effective then minimizer as it is only going to find additional workarounds. The largest difference will come from if the random algorithm was lucking in choosing configurations to test.

Looking at the results from the random algorithm the first test which has a chance to find all 30 of the bugs with workarounds is with 25 iterations which took on average 132 minutes. This will require some luck however as four of the bugs with workarounds found were not found during all five trial runs. The 100 iterations random multi minimization test found at least a workaround for all 30 bugs with workarounds on all 5 trial runs. This test took on average 805 minutes, still about 10 times less then the two-hop algorithm.

The two-way covering array algorithm takes only 46 minutes, slightly more than the one-hop algorithm. It only finds workarounds though for 25 of the 30 bugs with workarounds. To improve its efficiency we created five new two-way stacked covering arrays by combining two two-way covering arrays and ran the tests again. The two-way stacked covering arrays took 162 minutes and found workarounds for all 30 bugs with workarounds, though two of the bugs with workarounds were missed during one of the five trials. Next we tried a three way covering array algorithm and it can find workarounds for all 30 bugs with workarounds missing only one bug on one test. This algorithm took on average 186 minutes, making it slightly less efficient then the stacked two-way covering arrays.

The other comparison made is the time to first workaround only. If finding a single workaround is your top priority and the algorithm can stop once it has been discovered, then the average time spent should drop. Table 4.4 shows the results for the two-way covering array, three way covering array and the random algorithm. The random algorithm is the slowest taking on average almost 17 minutes. The two-way covering array takes slightly more then 15 minutes and the three way covering array takes about 12 and a half minutes. For these tests then, the three way covering array is the most efficient.

It is difficult to answer question three because of the randomness of the algorithms. The covering arrays are more systematic then random but still can't guarantee finding workarounds for all bugs with workarounds due to masking of features. Our results show that the stacked two-way covering array and three way covering array are comparable to random with about 50 iterations. The minimizer version of the 50 iterations random took 275 minutes which makes it much less efficient then either of the covering arrays. Because of this, and the fact that the covering arrays performed better at time to first workaround, we consider the covering arrays as the most efficient and effective method to finding workarounds for failures.

Chapter 5

Failure Avoidance Feasibility in Practice

In this chapter we present a study on an implementation of the failure avoidance framework. The goal is to evaluate if the framework will be successful in a real system running multiple distributed clients. We will measure this by determining if the workarounds found and the modifications made to the clients result in a drop of time to the first workaround and if we can completely avoid failures that have been seen.

We chose Firefox as the target system. Work by Garvin et al. [25] showed that Firefox contains configuration dependent bugs. This is a real highly-configurable application in which users are distributed and diverse. If we can get our framework to successfully run in this environment, then it will be a good indication if it can work in practice.

5.1 Simulation Setup

We built a simulation of the failure avoidance framework to represent four users of Firefox, working in parallel. We use Firefox version 18.0 [38] for the simulation. This was chosen because it is a recent version that is compatible with the Mozmill testing framework, and the Mozmill framework provides a way to automate the running of tests (Appendix B). The failure avoidance framework is built on top of Rainbow (as described in Chapter 3) that was provided to us by B. Schmerl at CMU [23].

To run the simulation, a master node is started first which loads the framework feature model and sets up the adaptation strategies. After this, the client nodes are activated, which involves deploying probes and creating a map between them and the properties on the master. This follows the standard implementation of Rainbow. Once all of the delegates are instantiated, the gauges activate and the monitoring begins. Because we use four clients to represent four Firefox users we will need to instantiate four clients in the framework. Each version of Firefox has its own version of Mozmill, which will select and run test cases; each test case is meant to simulate a user performing some task on the system.

Three critical parts of the simulation are as follows. The failure avoidance framework handles failures that the clients encounter by activating the failure avoiding algorithm and sending the results to the clients. Second, we need a Firefox feature model that defines the configuration space. Third, we need a Mozmill test suite that defines the tests to be run and a script to select which clients run which tests. Each of these aspects of the simulation will be further explained below. The simulation is set up and run on Sandhills. Sandhills is a cluster of 1440 AMD cores housed in a total of 44 nodes. This allows us to run multiple clients in parallel.

The simulation will differ from the real world in a few key areas. In the real world

the failure avoidance framework will need to be able to create test cases based on the behavior of a user leading up to a failure. In the simulation we use procreated mozmill tests. The real world will also have a much larger number of clients so the failure avoidance framework will need to handle a much larger number of simultaneous reported failures. The framework provides functionality for such an issue, however we leave the implementation as future work.

5.1.1 Failure Avoidance Framework

The failure avoidance framework as outlined in Chapter 3 is used in our simulation. We decided to use the Random, 10 iterations, single minimizer failure avoidance algorithm for this study. We chose this algorithm to start since it will run the fewest tests and performed relatively well in effectiveness. The Firefox feature model is significantly larger than the GCC feature model we used in our first study, therefore even the one-hop algorithm will be very expensive. The time required to find workarounds becomes important in this simulation, since we are now running this in real time. Because of this, and because we are going to stop at the first workaround, we won't use the multi-minimizer. Other variations such as using the covering array, adding the multi-minimizer, etc. are left as future work.

We have added one additional check done before the failure avoidance algorithm is called to search for a workaround. We want to make sure that any failure encountered is a real failure (and not just a transient failure due to the test harness failing), so the Mozmill test is rerun under the failing configuration before we begin our workaround search. If it does not fail, then the effector will return that the test was not reproducible to the client. If it does fail, then all of the already found workarounds are checked to see if they will avoid the failure. If one of the existing workarounds suc-

ceeds, then the effector will return that a workaround has already been discovered. Finally, if the test is reproducible and no workaround already exists, then the failure avoidance algorithm is called. If a workaround is discovered, the effector will return that a workaround has been found to the appropriate client, and then update all clients with the new guard information. If no workaround is found then the client is informed and regular testing can resume.

5.1.2 Failure Avoidance Algorithm

To set up the failure avoidance algorithm for our simulation it had to be modified to allow Mozmill tests to be run. The earlier tests were done using GCC bugs, but because the target system in the simulation is Firefox the algorithm will need to be able to run Mozmill test and determine wether it passed or failed under some configuration.

A script is used to call the Mozmill tests within the client. To adapt Mozmill testing into the failure avoidance algorithm we set up a driver to call this script. The failure avoidance algorithm can generate a configuration file for Firefox based on the current configuration in the test, and pass the location of the Mozmill file to be tested to the driver. The driver runs the Mozmill test in the same way as on the client, but this time using the given configuration from the failure avoidance algorithm.

Once the test is completed the failure avoidance algorithm can check the generated file with the results and determine if the test passed or failed. In this way the output running the test remains the same, and the failure avoidance algorithm is only minimally changed. This also means that not only can the algorithm used be swapped out, but as long as there is a way to automatically run a test case with a given configuration any system can be plugged in. This allows for customization of

the failure avoidance framework to a particular target system.

5.1.3 Firefox Feature Model

We created the feature model by obtaining a list of Firefox configurations and pruning it as described next. The starting list was the full list from the Firefox `about:config` utility. We first removed all of the hardware specific configuration options. Next we removed the security preferences assuming that a workaround that compromises security would be undesirable. We also removed preferences that dealt with plugins, extensions, and fonts. Finally we removed the more complicated String and Integer preferences leaving 311 preferences for us to work with. This is almost double the number of configurations used in the GCC study. The feature model then contains 639 options (due to some non binary preferences). This leads to a configuration space of approximately 1.30×10^{638}.

5.1.4 Firefox Mozmill Test Suite

Mozmill has a test suite that can be downloaded for use in regression testing of firefox [37]. We selected the functional category of tests from the most recent version of Firefox and ran all of them on one of our Firefox clients. Some of the tests did not pass, however this was expected since the version of Firefox being tested is older then the available Mozmill test suite. One example is that some of the tests have functions that perform differently in the version of Firefox being tested, and therefore will always fail.

After pruning out the failing tests, we are left with 36 Mozmill functional tests which always pass under the default Firefox configuration. By using these tests in the test suite we will be able to check that the configuration workarounds being found

won't overly impact the users functionality. If a workaround does cause a substantial drop in functionality related to one of the 36 tests, then the test that is supposed to pass will fail. In addition it is possible that a specific configuration will cause a passing test to fail, and if this is the case then the failure avoidance algorithm should be able to find a workaround. These will be categorized as unexpected workarounds within our simulation (discussed later).

We then added seven Mozmill tests we created to the test suite that represent configuration faults found in the Mozilla bug repository. The test will check to see if a bad configuration is selected in Firefox and then throw an error. This is done by creating an assert within a Mozmill test which checks the value of a configuration. An example of one of these tests is found in the appendix B.1. We looked at the Firefox bug repository [39] for faults reported in version 18 that involved configurations and if a failure was caused by a change in the configuration space it was turned into a Mozmill test. There were also three Firefox bugs from [25] that were found to be configuration dependent. If the failure could be replicated on Firefox 18 they were added as well. The full set of Mozmill tests can be found in the Appendix.

- Firefox bug 306208 − Firefox displays tab bar on pop ups when browser.tabs.drawinTitlebar is set to true (tab bar should only be displayed in main page), fails on changed configuration
- Firefox bug 344189 − Firefox opens 2 tabs when *browser.search.openintab* is set to true (only 1 tab should open), fails on default configuration
- Firefox bug 442970 − Firefox opens last session and home page when *browser.startup.page* is set to 3 (only last session should open), fails on changed configuration
- Firefox bug 505548 − Firefox gives no warning when *browser.startup.page* is set

to 3 (open previous session on startup) and *privacy.sanitize.sanitizeOnShutdown* is set to true (wipe all session data on closing), fails on changed configuration

- Firefox bug 797945 − Firefox doesn't apply binding properly when *plugins.click_to_play* is set to true, fails on default configuration

- Firefox bug 808290 − Firefox is unpredictable when *browser.link.open_newwindow* is set to 1 and a link is clicked immediately before focusing on a new tab, fails on changed configuration

- Firefox bug 840411 − about:home won't work on firefox if *browser.startup.homepage_override.* is set to "ignore", fails on changed configuration

These bugs were then encoded as Mozmill tests which will check the current Firefox client's configuration and throw a failure if the bad configuration is detected. This gives us an accurate way of recreate the failures without having to worry about changes to Firefox. The failure avoidance algorithm should be able to find a workaround for each of these tests when they fail by minimizing out of the configuration option that caused the test to fail. These will be categorized as expected workarounds within our simulation.

Overall we have 36 passing tests and seven tests designed to fail for our simulation. The designed to fail tests however, might require a reconfiguration to fail, which means the likelihood of encountering them is lower.

5.1.5 Simulating the Clients

The users are simulated by Algorithm 6 that randomly calls a test from our Mozmill test suite. Each client keeps track of its current configuration, the guard information and if a failure has been encountered. The overall idea is that the clients will explore

Algorithm 6 Algorithm for Running Client

```
let M ← the suite of mozmill tests
let c ← the current configuration
let r ← a single option from c
let G ← the set of Guarded reconfigurations
let f ← false (represents if a failure has not been discovered)
while Client is running do
    if f = false (a failure has not been discovered) then
        Change the configuration c by up to one reconfiguration r unless r ∈ G
        Randomly select a mozmill test m ∈ M and run m.
        if m fails under configuration c then
            let f ← true
        end if
    end if
    if An update for the guard g is available then
        add g to G
    end if
    if A workaround configuration c' has been discovered for test m then
        note that c' is a workaround for m
        let f ← false
    else if The test m failing under configuration c could not be replicated then
        note that m with c is not replicable
        let f ← false
    else if A workaround could not be found for test m failing under configuration
c then
        note that m with c has no current workaround
        let f ← false
    end if
end while
```

the configuration space while trying different mozmill tests. When a failure is encountered the client will stop running new Mozmill tests, and wait for a response from the failure avoidance framework effector. The response will either be that a workaround is found, a workaround is not found, or the test cannot be replicated. In all cases once a response has been received the client can resume running tests. In addition the client will be checking to see if the guard can be updated with a new potentially bad reconfiguration.

This set up allows the system to keep track not only of the failing test case, but also the configuration at the time the test failed. This will allow the failure to be replicated and make it possible to search for a workaround. When a failure is encountered the client will stop until a message has been sent from the effector. It is possible for a user to continue working once a failure has been encountered, however for the purpose of this simulation we wanted to keep track of time between detecting a failure and getting a response without a bottleneck on the queue. We are only using one master and one failure avoidance algorithm so failures can only be addressed one at a time in the order they are received.

The starting configuration is kept as default for one of the users, and varies by up to five configuration options for the other three. Before each test is run there is also a chance that the configuration will change by one configuration option. This is where the guard algorithm is implemented. As workarounds are found the configuration information is deployed to each of the clients, which allows them to update their own personal guard. Before each Mozmill test is run the client checks its current configuration, and has a chance to randomly change a single configuration option. This number was kept low so that the exploration of the configuration space didn't expand too rapidly. We thought this would be a more realistic simulation of users who are unlikely to move far away from the default configuration quickly, but who may creep away over time. Before each reconfiguration the client checks with the guard for permission to make the change.

5.1.5.1 Use of the Guard

As stated earlier, each time a Mozmill test is run, the client has a chance to change its current configuration. At the time of this change the client will also check to make sure that the change will not bring the configuration to a potentially bad state.

This ensures that after a workaround is found, the failure should not be encountered again. It also provides an easy way to spread workaround knowledge to all clients. By keeping the guard updated a client ensures that it will not encounter a failure that another client has already found a workaround for.

If two clients both encounter a failure at around the same time, then it is possible a queue will form while the failure avoidance algorithm is working. In this case it is possible that a workaround found for one of the clients will also serve as a workaround for the other. This means that the failure was encountered before the guard had a chance to be updated. The second client does not know the workaround it has received will avoid the failure it sent to the master. The algorithm must therefore check to see if a found workaround can serve for the failure it is looking at, and if it is, let the client know that their guard is up to date and they can continue running tests.

This situation will result in a case where a client is being protected from a failure by the guard, but is still waiting for a response from the master. The response time for this case will be evaluated further in the case study.

5.1.6 Record Keeping

While the simulation is running each of the clients will keep track of tests run, updates to the guard, guard activation, and results sent back from the master. All of these will be be marked with a timestamp so comparisons can be made between the clients. An example of this is stored in the appendix C.1. In addition after each Mozmill test is run a file will be created that contains the current configuration so that tests can be replicated. The master will keep track of workarounds, and tests run with the result of each. By compiling this information we will be able to accurately look at the simulation results and determine if the current system is working as intended.

5.2 Case Study

We conducted a case study to determine if the failure avoidance framework works. We first ask if the framework can find workarounds of all the expected workarounds in the test suite as well as is the guard working to prevent failures with found workarounds from being encountered again. In this study we answer two questions.

RQ1. Can the failure avoidance framework efficiently find workarounds for all of the failing test cases we introduced to the test suite?

RQ2. Is the guard effective at preventing future failures once a workaround has been found?

Objects of Study. Our evaluation of the failure avoidance framework was done using Firefox version 18 [38] as the target system and the Random with 10 iterations and the minimizer as the failure avoidance algorithm. The Feature model for firefox was constructed using the `about:config` configurations from Firefox and pruned as described above. The test suite for Firefox was created by combining a subsection of the default Mozmill test suite for functional tests found at [37] and the Mozmill tests from [39] and [25] designed to fail under certain configurations described above.

5.2.1 Independent and Dependent Variables

The independent variable in our study is the time the simulation will run.

Our dependent variables are the number of workarounds and the number of guards activated.

Test	0:15	0:30	0:45	1:00	1:15	1:30	1:45	2:00	2:15	2:30	2:45	3:00
Expected to Fail												
firefoxBug_344189.js	FW	2P	4P	5P	-	PG	4P	2P	8P	G7PG3P	3P	4PG2P
firefoxBug_306208.js	P	2P	P	5P	5P	3P	3P	4P	2P	P	5P	5P
firefoxBug_797945.js	-	FWP	3P2GP	2P	3P	5P	3P	5P	2P	6P	2PGP	PG5PG
firefoxBug_808290.js	P	3P	P	4P	5P	P	P	3P	4P	4P	6P	P
firefoxBug_840411.js	2P	4P	2P	3P	-	4P	6P	3P	8P	4P	2P	6P
firefoxBug_442970.js	P	2P	4P	P	4P	P	3P	2P	5P	2P	2P	5P
firefoxBug_505548.js	7P	7P	-	4P	5P	4P	4P	3P	2P	6P	6P	3P
Not Expected to Fail												
testClearFormHistory.js	4P	3P	4P	3P	FPW5P	3P	P	5P	P	3PG	4P	GP
testPopupsBlocked.js	5P	3P	P	4P	P	PF2PWP	6P	3P	5P	6P	2P	5P
testAddMozSearchProvider.js	3P	3PF	PW2P	P	4P	4P	7P	6P	5P	4P	4P	3P
Does Not Fail												
testGoButton.js	5P	4P	P	5P	4P	5P	2P	2P	P	4P	3P	4P
testLocationBarSearches.js	5P	4P	5P	5P	3P	2P	5P	3P	5P	6P	2P	2P
testPasteLocationBar.js	5P	P	5P	7P	5P	3P	6P	2P	5P	2P	11P	3P
testAddBookmarkToMenu.js	2P	2P	7P	3P	4P	P	3P	5P	7P	10P	3P	6P
testCloseDownloadManager.js	5P	2P	6P	P	P	6P	2P	9P	2P	3P	6P	3P
testOpenDownloadManager.js	P	5P	2P	5P	4P	2P	4P	5P	4P	2P	2P	3P
testAutoCompleteOff.js	6P	6P	5P	5P	4P	-	4P	6P	3P	4P	3P	4P
testBasicFormCompletion.js	4P	9P	7P	10P	4P	6P	6P	2P	3P	4P	4P	6P
testDisableFormManager.js	-	2P	4P	6P	P	4P	5P	2P	4P	3P	3P	5P
testNavigateFTP.js	6P	P	10P	5P	5P	2P	2P	3P	4P	4P	2P	4P
testPopupsAllowed.js	3P	4P	3P	6P	P	4P	2P	3P	5P	5P	7P	5P
testDefaultPhishingEnabled.js	4P	6P	4P	4P	8P	3P	2P	6P	5P	5P	3P	7P
testDefaultSecurityPrefs.js	4P	7P	9P	7P	3P	2P	3P	P	5P	4P	8P	7P
testPaneRetention.js	4P	7P	4P	3P	-	2P	3P	2P	4P	5P	7P	3P
testPasswordNotSaved.js	5P	4P	2P	2P	6P	4P	3P	7P	7P	3P	6P	4P
testPreferredLanguage.js	3P	3P	3P	4P	3P	4P	3P	4P	4P	5P	4P	5P
testRestoreHomepageToDefault.js	4P	P	5P	4P	3P	P	3P	6P	8P	5P	3P	3P
testSetToCurrentPage.js	3P	P	6P	3P	5P	4P	2P	7P	3P	3P	7P	3P
testSwitchPanes.js	7P	P	7P	6P	3P	P	6P	8P	4P	3P	2P	5P
testOpenSearchAutodiscovery.js	7P	-	4P	P	4P	4P	6P	6P	6P	5P	6P	3P
testSearchViaFocus.js	2P	-	-	P	P	-	3P	2P	4P	-	4P	P
testSearchViaShortcut.js	4P	-	4P	2P	3P	4P	8P	5P	4P	5P	8P	4P
testDVCertificate.js	2P	3P	5P	2P	2P	P	3P	6P	7P	4P	4P	4P
testEnablePrivilege.js	3P	5P	5P	5P	3P	5P	5P	5P	3P	7P	P	5P
testIdentityPopupOpenClose.js	P	P	3P	7P	P	P	5P	6P	P	4P	6P	7P
testSSLDisabledErrorPage.js	9P	2P	5P	7P	3P	5P	6P	5P	6P	5P	3P	5P
testSubmitUnencryptedInfoWarning.js	2P	9P	2P	5P	4P	2P	5P	3P	4P	5P	3P	6P
testUntrustedConnectionErrorPage.js	4P	P	5P	6P	4P	4P	5P	2P	P	4P	3P	5P
testCloseTab.js	2P	4P	2P	5P	2P	3P	9P	3P	3P	6P	7P	3P
testTabGroupNaming.js	3P	3P	4P	6P	5P	-	2P	4P	3P	5P	P	3P
testToggleTabView.js	4P	P	5P	3P	2P	2P	5P	6P	5P	3P	3P	6P
testBackForwardButtons.js	2P	2P	P	2P	6P	6P	6P	3P	5P	3P	5P	4P
testHomeButton.js	4P	6P	3P	5P	2P	5P	5P	6P	4P	8P	P	6P

Table 5.1: Simulation of System Run 1

5.2.2 Simulation Results

Five trails of the simulation were run, each one lasting at least three hours. The results of these runs are recorded in tables 5.1 through 5.5. The tables show 15 minute time slots. A P represents a passing test case for one of the clients during that time period. F represents a failing test case for one of the clients. W means a

Test	0:15	0:30	0:45	1:00	1:15	1:30	1:45	2:00	2:15	2:30	2:45	3:00
Expected to Fail												
firefoxBug_344189.js	FWGP	G2P	7P	4P	G4P	G2P	2P	2PG2P	2P	2P	2PG	2P
firefoxBug_306208.js	3P	3P	4P	3P	P	5P	P	2P	4P	2P	4P	P
firefoxBug_797945.js	F	W2P	3P	4P	2PG2P	2P	6P	3P	P	4P	5P	3P
firefoxBug_808290.js	3P	–	P	P	4P	P	P	2P	2P	6P	7P	3PF
firefoxBug_840411.js	5P	4P	2P	–	3P	4P	5P	5P	PFWP	2P	2PGP	G
firefoxBug_442970.js	3P	5P	6P	P	2P	3P	4P	–	3P	5P	4P	2P
firefoxBug_505548.js	4P	2P	2P	3P	2P	2P	3P	2P	3P	5P	3P	3P
Not Expected to Fail												
testPopupsAllowed.js	P	P	5P	FP	PW2P	PGP	G5P	4PG3P	2P	3P	5P	3P
Does Not Fail												
testGoButton.js	P	5P	2P	6P	4P	–	3P	2P	4P	2P	4P	P
testLocationBarSearches.js	4P	2P	2P	P	6P	2P	7P	2P	3P	2P	2P	–
testPasteLocationBar.js	3P	4P	–	2P	4P	3P	–	P	2P	3P	P	2P
testAddBookmarkToMenu.js	4P	3P	2P	4P	4P	6P	5P	4P	3P	2P	–	2P
testCloseDownloadManager.js	4P	2P	5P	3P	3P	8P	2P	3P	5P	3P	5P	P
testOpenDownloadManager.js	2P	3P	5P	P	2P	4P	4P	5P	3P	4P	4P	4P
testAutoCompleteOff.js	4P	2P	3P	P	6P	9P	P	P	5P	4P	3P	2P
testBasicFormCompletion.js	3P	4P	3P	P	P	4P	3P	P	5P	3P	4P	4P
testClearFormHistory.js	2P	5P	5P	2P	–	2P	3P	5P	3P	2P	2P	–
testDisableFormManager.js	3P	3P	8P	6P	3P	4P	2P	7P	P	2P	PF	P
testNavigateFTP.js	3P	2P	7P	–	P	4P	6P	2P	3P	5P	3P	4P
testPopupsBlocked.js	P	2P	5P	–	P	2P	3P	4P	6P	4P	3P	P
testDefaultPhishingEnabled.js	2P	4P	3P	6P	6P	5P	7P	4P	3P	–	3P	2P
testDefaultSecurityPrefs.js	2P	2P	2P	4P	P	P	P	6P	2P	5P	2P	4P
testPaneRetention.js	5P	P	P	4P	3P	6P	2P	4P	P	3P	2P	6P
testPasswordNotSaved.js	2P	P	5P	8P	3P	4P	4P	4P	4P	5P	–	2P
testPreferredLanguage.js	2P	3P	3P	2P	3P	3P	3P	4P	5P	2P	4P	4P
testRestoreHomepageToDefault.js	2P	2P	–	2P	2P	4P	5P	P	3P	4P	3P	2P
testSetToCurrentPage.js	P	4P	3P	6P	3P	4P	2P	2P	4P	4P	7P	2P
testSwitchPanes.js	4P	P	2P	–	6P	P	P	P	2P	5P	P	–
testAddMozSearchProvider.js	4P	6P	6P	2P	4P	6P	8P	7P	–	7P	2P	–
testOpenSearchAutodiscovery.js	7P	3P	5P	P	4P	2P	5P	2P	3P	3P	5P	4P
testSearchViaFocus.js	2P	3P	3P	P	3P	P	3P	2P	3P	P	3P	3P
testSearchViaShortcut.js	4P	P	5P	3P	3P	2P	3P	3P	3P	3P	3P	3P
testDVCertificate.js	3P	3P	2P	4P	8P	3P	4P	2P	P	5P	5P	P
testEnablePrivilege.js	–	2P	4P	6P	2P	3P	3P	5P	2P	6P	4P	P
testIdentityPopupOpenClose.js	2P	2P	2P	P	3P	9P	5P	2P	3P	–	4P	4P
testSSLDisabledErrorPage.js	3P	4P	3P	3P	3P	3P	3P	3P	–	4P	7P	P
testSubmitUnencryptedInfoWarning.js	2P	7P	2P	2P	4P	–	2P	P	2P	4P	4P	3P
testUntrustedConnectionErrorPage.js	–	6P	2P	P	3P	5P	2P	4P	P	2P	2P	3P
testCloseTab.js	2P	2P	4P	2P	3P	2P	2P	2P	5P	3P	2P	2P
testTabGroupNaming.js	2P	P	7P	–	4P	5P	P	4P	4P	3P	–	P
testToggleTabView.js	–	3P	7P	3P	2P	3P	2P	2P	–	3P	5P	–
testBackForwardButtons.js	P	2P	2P	3P	3P	3P	3P	–	3P	6P	4P	2P
testHomeButton.js	4P	3P	3P	P	2P	7P	4P	6P	P	4P	P	–

Table 5.2: Simulation of System Run 2

Test	0:15	0:30	0:45	1:00	1:15	1:30	1:45	2:00	2:15	2:30	2:45	3:00
Expected to Fail												
firefoxBug_344189.js	FW	2P	G2P2G	3P	7P	6P	4P	8P	7P	P	P	3P
firefoxBug_306208.js	P	6P	P	7P	6P	2P	P	4P	4P	P	3P	2P
firefoxBug_797945.js	FFWA	5P	4P	3P	5P	5P	6PG	4P	4P	2PGP	5P	PG3P
firefoxBug_808290.js	2P	4P	4P	5P	–	4P	2P	6P	3P	6P	5P	2PFPWP
firefoxBug_840411.js	5P	FPW7P	5P	4P	2PGPG4P	8P	6PGP	2PGP	3P	3P	4P	2PGP
firefoxBug_442970.js	3P	6P	P	4P	6P	6P	6P	4P	2P	PF2P	NFPW	7P
firefoxBug_505548.js	6P	2P	3P	9P	5P	5P	P	4P	5P	2P	4P	2P
Not Expected to Fail												
testPopupsAllowed.js	4P	6P	4P	5P	5P	P	7P	5PF	PW3P	8P	P	2P
Does Not Fail												
testGoButton.js	5P	6P	3P	6P	5P	4P	2P	5P	2P	P	–	2P
testLocationBarSearches.js	5P	4P	6P	4P	2P	7P	4P	2P	5P	2P	5P	3P
testPasteLocationBar.js	P	3P	5P	4P	2P	7P	9P	3P	3P	4P	3P	10P
testAddBookmarkToMenu.js	4P	3P	5P	5P	5P	4P	6P	3P	3P	P	P	5P
testCloseDownloadManager.js	4P	4P	3P	9P	P	4P	8P	2P	4P	2P	3P	4P
testOpenDownloadManager.js	P	3P	5P	P	4P	P	4P	4P	4P	4P	6P	P
testAutoCompleteOff.js	4P	4P	2P	–	5P	5P	2P	6P	3P	2P	6P	6P
testBasicFormCompletion.js	P	7P	7P	4P	2P	5P	4P	4P	5P	P	2P	2P
testClearFormHistory.js	4P	2P	3P	4P	8P	3P	5P	2P	7P	4P	2P	6P
testDisableFormManager.js	4P	4P	4P	4P	8P	5P	5P	3P	4P	2P	9P	5P
testNavigateFTP.js	6P	3P	–	–	5P	3P	3P	2P	6P	4P	3P	6P
testPopupsBlocked.js	2P	4P	4P	7P	5P	5P	7P	2P	3P	4P	4P	3P
testDefaultPhishingEnabled.js	3P	4P	4P	4P	4P	3P	4P	2P	9P	4P	5P	P
testDefaultSecurityPrefs.js	3P	3P	8P	5P	5P	3P	6P	5P	2P	P	P	5P
testPaneRetention.js	P	5P	2P	3P	5P	5P	5P	6P	5P	4P	5P	2P
testPasswordNotSaved.js	–	4P	2P	7P	4P	3P	2P	3P	3P	2P	3P	6P
testPreferredLanguage.js	4P	2P	7P	9P	2P	3P	2P	4P	P	4P	3P	–
testRestoreHomepageToDefault.js	3P	8P	4P	3P	4P	4P	4P	7P	2P	–	3P	7P
testSetToCurrentPage.js	4P	4P	5P	3P	4P	4P	6P	5P	6P	5P	P	3P
testSwitchPanes.js	3P	3P	4P	13P	2P	2P	4P	4P	5P	2P	2P	4P
testAddMozSearchProvider.js	3P	6P	P	4P	6P	3P	4P	P	3P	4P	7P	6P
testOpenSearchAutodiscovery.js	6P	4P	3P	–	8P	8P	3P	5P	5P	2P	7P	P
testSearchViaFocus.js	2P	–	P	–	2P	2P	5P	2P	3P	3P	–	2P
testSearchViaShortcut.js	–	4P	4P	3P	7P	5P	9P	7P	5P	2P	7P	3P
testDVCertificate.js	6P	6P	7P	5P	P	4P	5P	2P	5P	6P	8P	–
testEnablePrivilege.js	P	2P	5P	3P	4P	2P	2P	3P	4P	5P	6P	3P
testIdentityPopupOpenClose.js	3P	3P	6P	3P	3P	7P	2P	6P	3P	P	2P	3P
testSSLDisabledErrorPage.js	6P	4P	5P	2P	6P	6P	2P	8P	6P	2P	4P	2P
testSubmitUnencryptedInfoWarning.js	P	4P	5P	3P	8P	3P	P	4P	P	6P	10P	4P
testUntrustedConnectionErrorPage.js	3P	5P	7P	4P	7P	3P	3P	6P	3P	P	7P	6P
testCloseTab.js	4P	4P	5P	6P	P	6P	2P	5P	6P	3P	5P	6P
testTabGroupNaming.js	P	5P	9P	2P	2P	5P	6P	5P	4P	4P	3P	5P
testToggleTabView.js	P	4P	3P	6P	3P	2P	3P	3P	3P	3P	P	4P
testBackForwardButtons.js	2P	4P	P	5P	3P	6P	3P	5P	3P	2P	4P	3P
testHomeButton.js	2P	4P	5P	2P	P	4P	5P	3P	2P	3P	2P	4P

Table 5.3: Simulation of System Run 3

Test	0:15	0:30	0:45	1:00	1:15	1:30	1:45	2:00	2:15	2:30	2:45	3:00
Expected to Fail												
firefoxBug_344189.js	FWP	G5P	3P	5P	7P	4P	G4P	3P	6P	4P	4P	6P
firefoxBug_306208.js	P	3P	5P	6P	5P	8P	PF	PW3P	P	4P	4P	2P
firefoxBug_797945.js	2FWA	PG4P	4PG	6PG2PG	6P	2G2P	5P	2PG	2PG2PG	6P	4P	2P
firefoxBug_808290.js	P	2P	6P	P	4P	5P	2P	4P	P	3P	5P	2P
firefoxBug_840411.js	2P	4P	3P	5P	2P	9P	2P	2P	3P	3P	3P	4P
firefoxBug_442970.js	4P	3P	2P	3P	6P	5P	3P	5P	P	3P	–	3P
firefoxBug_505548.js	P	3P	2P	5P	3P	4P	–	4P	P	6P	–	7P
Not Expected to Fail												
testOpenSearchAutodiscovery.js	P	3P	3P	7P	–	4P	3P	3P	3P	2PF2P	W5P	3P
testClearFormHistory.js	P	5P	3P	4P	5P	5P	5P	2PF	WP	3P	3P	6P
testAutoCompleteOff.js	5P	3PF	PN2P	2P	4P	P	4P	5P	5P	4P	5P	4P
Does Not Fail												
testGoButton.js	P	3P	4P	4P	P	4P	5P	2P	3P	4P	3P	2P
testLocationBarSearches.js	P	4P	4P	9P	3P	3P	5P	2P	–	2P	7P	8P
testPasteLocationBar.js	2P	3P	6P	3P	2P	3P	4P	3P	6P	7P	2P	4P
testAddBookmarkToMenu.js	–	4P	2P	3P	P	4P	3P	2P	4P	2P	6P	5P
testCloseDownloadManager.js	P	P	5P	2P	3P	3P	4P	3P	3P	3P	4P	5P
testOpenDownloadManager.js	P	3P	5P	P	P	4P	2P	4P	3P	7P	3P	4P
testBasicFormCompletion.js	P	3P	3P	2P	5P	P	P	3P	P	2P	7P	5P
testDisableFormManager.js	–	P	3P	5P	4P	P	2P	P	2P	2P	2P	3P
testNavigateFTP.js	P	2P	5P	P	3P	4P	3P	3P	4P	9P	3P	6P
testPopupsAllowed.js	2P	6P	6P	4P	6P	4P	3P	–	5P	5P	3P	3P
testPopupsBlocked.js	3P	5P	2P	3P	6P	2P	2P	P	2P	2P	3P	–
testDefaultPhishingEnabled.js	–	5P	6P	4P	2P	3P	3P	4P	4P	P	6P	2P
testDefaultSecurityPrefs.js	2P	P	3P	3P	3P	4P	4P	2P	3P	4P	5P	6P
testPaneRetention.js	P	2P	P	3P	9P	4P	2P	2P	4P	2P	2P	3P
testPasswordNotSaved.js	–	5P	4P	6P	3P	2P	P	3P	3P	P	2P	5P
testPreferredLanguage.js	P	7P	5P	6P	4P	4P	3P	P	P	2P	3P	–
testRestoreHomepageToDefault.js	P	7P	2P	2P	4P	4P	4P	5P	3P	P	4P	4P
testSetToCurrentPage.js	2P	3P	5P	6P	2P	2P	7P	4P	3P	2P	2P	2P
testSwitchPanes.js	–	3P	4P	5P	4P	5P	–	2P	4P	3P	3P	7P
testAddMozSearchProvider.js	3P	8P	5P	4P	3P	3P	–	4P	4P	2P	5P	3P
testSearchViaFocus.js	P	4P	P	2P	3P	P	P	2P	4P	2P	–	2P
testSearchViaShortcut.js	2P	P	5P	5P	6P	4P	2P	P	P	P	2P	2P
testDVCertificate.js	–	4P	4P	3P	4P	3P	3P	5P	3P	2P	3P	2P
testEnablePrivilege.js	2P	9P	4P	3P	5P	6P	4P	3P	7P	3P	3P	4P
testIdentityPopupOpenClose.js	2P	6P	3P	3P	3P	3P	P	–	2P	P	5P	4P
testSSLDisabledErrorPage.js	–	4P	7P	–	5P	3P	2P	3P	5P	P	6P	P
testSubmitUnencryptedInfoWarning.js	–	P	7P	7P	3P	5P	–	–	2P	2P	6P	6P
testUntrustedConnectionErrorPage.js	–	7P	4P	5P	2P	5P	P	6P	6P	3P	3P	3P
testCloseTab.js	P	7P	3P	4P	6P	3P	3P	P	4P	2P	3P	P
testTabGroupNaming.js	–	8P	7P	5P	2P	3P	5P	2P	7P	4P	3P	4P
testToggleTabView.js	2P	5P	2P	3P	2P	8P	4P	2P	3P	P	–	4P
testBackForwardButtons.js	3P	6P	4P	4P	8P	6P	2P	2P	–	2P	3P	9P
testHomeButton.js	P	5P	6P	6P	4P	5P	P	P	6P	3P	2P	2P

Table 5.4: Simulation of System Run 4

Test	0:15	0:30	0:45	1:00	1:15	1:30	1:45	2:00	2:15	2:30	2:45	3:00
Expected to Fail												
firefoxBug_344189.js	FW	2P	2P	P	P	2P	3P	3P	G	3P	5P	3P
firefoxBug_306208.js	F	FWAPG	3P	P	PG	4P	4P	4P	2P	PG	2P	6P
firefoxBug_797945.js	FWPG	G	5P	3P	4P	2P	2P	2P	4P	3P	3P	PG
firefoxBug_808290.js	–	P	P	2P	P	4P	P	PFW	2P	3P	P	–
firefoxBug_840411.js	3P	2P	5P	FPW2P	3P	–	6P	P	PG	–	3P	PG
firefoxBug_442970.js	3P	–	2P	P	2P	2P	2P	2P	P	3P	2P	4P
firefoxBug_505548.js	–	3P	4P	3P	4P	2P	–	P	2P	2P	P	2P
Not Expected to Fail												
testOpenSearchAutodiscovery.js	P	P	6P	5P	3P	2P	3P	2P	5P	PF	W3P	2P
testSSLDisabledErrorPage.js	FP	PA	–	2P	3P	2P	2P	5P	4P	P	3P	3P
testUntrustedConnectionErrorPage.js	FN	P	3P	4P	3P	P	2P	P	2P	3P	2P	2P
Does Not Fail												
testGoButton.js	P	2P	3P	3P	2P	2P	2P	4P	P	–	P	3P
testLocationBarSearches.js	P	2P	6P	3P	3P	2P	3P	–	P	–	2P	–
testPasteLocationBar.js	P	2P	4P	3P	3P	–	3P	2P	3P	2P	P	P
testAddBookmarkToMenu.js	–	2P	6P	4P	3P	2P	P	4P	–	P	2P	P
testCloseDownloadManager.js	P	3P	3P	–	2P	4P	P	2P	3P	5P	–	3P
testOpenDownloadManager.js	P	2P	4P	3P	2P	P	3P	P	4P	–	2P	4P
testAutoCompleteOff.js	–	2P	3P	5P	2P	2P	–	2P	3P	P	3P	3P
testBasicFormCompletion.js	P	P	5P	2P	P	6P	P	6P	3P	3P	P	6P
testClearFormHistory.js	2P	–	3P	2P	2P	3P	P	P	5P	P	2P	P
testDisableFormManager.js	P	3P	P	2P	2P	5P	P	P	2P	5P	–	P
testNavigateFTP.js	3P	4P	2P	4P	–	P	2P	–	5P	P	P	2P
testPopupsAllowed.js	P	P	–	P	2P	4P	3P	P	3P	–	P	3P
testPopupsBlocked.js	–	3P	2P	3P	3P	5P	3P	2P	2P	2P	P	P
testDefaultPhishingEnabled.js	P	P	4P	2P	2P	3P	3P	–	2P	3P	–	3P
testDefaultSecurityPrefs.js	P	P	2P	2P	P	2P	–	P	3P	–	P	6P
testPaneRetention.js	P	2P	2P	3P	2P	3P	P	P	4P	2P	3P	3P
testPasswordNotSaved.js	3P	2P	3P	3P	3P	3P	4P	P	3P	2P	5P	P
testPreferredLanguage.js	–	3P	4P	–	–	–	2P	3P	2P	3P	2P	–
testRestoreHomepageToDefault.js	3P	3P	2P	4P	P	4P	2P	P	P	P	3P	–
testSetToCurrentPage.js	P	2P	2P	3P	P	5P	5P	3P	P	3P	3P	3P
testSwitchPanes.js	–	3P	3P	2P	–	2P	4P	P	2P	2P	2P	3P
testAddMozSearchProvider.js	P	P	2P	2P	P	2P	4P	P	3P	2P	–	5P
testSearchViaFocus.js	P	–	–	–	–	–	–	–	–	–	–	–
testSearchViaShortcut.js	P	3P	4P	4P	3P	3P	3P	2P	P	2P	–	2P
testDVCertificate.js	2P	2P	6P	3P	–	2P	–	2P	3P	5P	2P	P
testEnablePrivilege.js	–	3P	2P	P	5P	6P	2P	3P	P	3P	2P	3P
testIdentityPopupOpenClose.js	P	4P	7P	3P	–	4P	P	P	2P	3P	6P	2P
testSubmitUnencryptedInfoWarning.js	–	2P	8P	2P	3P	5P	P	P	–	2P	3P	P
testCloseTab.js	3P	–	2P	P	4P	4P	P	–	3P	2P	P	P
testTabGroupNaming.js	P	2P	P	4P	–	P	–	3P	2P	P	3P	2P
testToggleTabView.js	P	2P	4P	2P	6P	3P	2P	P	4P	P	P	2P
testBackForwardButtons.js	P	P	P	2P	2P	P	3P	3P	P	P	4P	–
testHomeButton.js	P	4P	3P	2P	2P	P	3P	P	2P	P	–	–

Table 5.5: Simulation of System Run 5

workaround was discovered, A means a workaround had already been discovered, and N means a workaround was not discovered. A number in front of the letter indicates how many times this even occured (e.g. $5P$ means a test ran and passed 5 times in that time slot). The first section of the table presents the results from the Mozmill we added to the test suite. These are tests that we expect to fail at some point during the simulation. The second section presents Mozmill test cases that were not expected to fail, but did (and were reproducible). Finally the last section is made up of the Mozmill tests that were not expected to and did not fail.

Bug 344189 and bug 797945 fail with the default configuration. It is possible that the clients will mutate to a safe configuration for these bugs randomly, but the odds are that the first time the bugs are encountered they will fail, and the search for a workaround will begin. This is what we see from the simulation results as the first time the bugs are seen the symbol is F. There is only one other time when a bug fails the first time it is encountered and that is on system run 5. Bug 306208 fails the first two times it is seen. Investigation into this shows that two of the clients were simply unlucky with their pick of starting configuration.

The other bugs seem to be hit randomly throughout the simulation, which is what we would expect. As changes are made to the configuration each time a test is called the likelihood of mutating into an unsafe state and then calling a failing test case increases. However, the system works as designed by detecting the failure, finding the workaround, and then deploying it to all clients so that the guard can be used to avoid hitting the failure in the future. We see this as once a W has been seen for a simulation there are no cases where a F is then seen.

5.2.2.1 Expected Workarounds

Of the seven Firefox bugs over the course of the five, three hour simulation runs, workarounds were found for six of the bugs. No workaround was ever found for bug 505548, however this can be explained by looking at bug 442970. A workaround for bug 442970 will create a guard to prevent a client from setting *browser.startup.page* to 3. Because *browser.startup.page* set to 3 is one of the two configuration options necessary to cause bug 505548 to fail, the test cannot fail once a workaround for bug 442970 is found.

The system was double checked by forcing a client to only test bug 505548. After running for a while it eventually saw the failure, and the failure avoidance algorithm was able to find a workaround for the failure. Because the two options necessary are both reconfigurations from the starting configuration it makes sense that the likelihood of hitting the failure for bug 505548 is much less then bug 442970. This does however show that easier to find failures can result in workarounds that guard against harder to discover failures and confers with the locality result of Garvin et al. [25].

Obviously the failures that require a default configuration option are discovered first. In every simulation at least one of the clients will test the bug before reconfiguring to a safe state. For the other Firefox bugs it is random. Not only does a client have to be in the right configuration, but it then has to call the right test to see the failure. It is entirely possible that a client configures to a dangerous state, and then reconfigures out of it before the test is called. Despite this at least one of the non default failing test cases is encountered in each of the three hour simulations.

In simulation three, shown in Table 5.3 a failure for bug 442970 is encountered, but no workaround is found. Later on, however, that failure is encountered and

a workaround is found. The failure avoidance algorithm is using the 10 iteration random with a minimizer that stops after a workaround is found. From our earlier gcc tests we know there is a chance that each of the randomly chosen configurations might not contain the necessary reconfiguration for a workaround. This shows that the random algorithm cannot be guaranteed to succeed, and gives evidence for using a more systematic algorithm, such as a covering array, in the future.

5.2.2.2 Unexpected Workarounds

From the beginning of building the simulation we were expecting to see failures other than those planned. Running on sandhills adds an extra level of complexity and more ways for things to go wrong. To start with we deal with the tests that fail and can't be replicated. These we would actually expect to see in a real system as not every failure will be replicable. There are also Mozmill failures that occur during test teardown.

If a Mozmill test fails and the failure is not replicable then the reason for the failure remains unknown. It is possible that the unknown failure could occur during the replication test, which would make the failure avoidance algorithm think the test can be replicated and a workaround should be searched for. Now if a random configuration is generated and the test is run it passes, the failure avoidance algorithm is going to think it has found a workaround and minimize it. The minimal workaround is just going to be whatever the delta algorithm minimizes to as the test should pass on any of the configurations.

This leads to a workaround that is deployed, but isn't really a workaround and the guard is now blocking what should be a safe configuration. We knew it would be possible to see these workaround false positives, especially as the length of running the simulation increases. Checking them at the end of running the simulation has lead to interesting results. Not all of them are false positives.

The Mozmill test *testFormManager/testClearFormHistory.js* failed during simulation run one shown in Table 5.1. It was replicable, and a workaround was found. To determine if it was a false positive we used the failing configuration given by the client and reran the test. The result was a failure, so it was at least replicable. The next step was to apply the workaround and test the result. The workaround found was the configuration option *browser.formfill.enable* is set to true. Rerunning the test with the workaround applied avoided the failure. This is a success for our system, as we were not expecting to find workarounds for failures that we did not put into the test suite. We knew it was a possibility, but we found that the failure avoidance works not only for failures found in the Mozilla bug repository, but also unexpected failures in the Mozmill test suite.

Another Mozmill test *testPopupsBlocked.js* also failed in simulation one and had a workaround found. The workaround for that failing test case is *javascript.enabled* set to true. Like the previous test case this can be verified, and as an added bonus the same failure and workaround is detected on multiple simulation runs. Changing *javascript.enabled* to false will cause *testPopupsBlocked.js* to fail.

As expected not all workarounds found were true workarounds. For the long simulation run, Mozmill test *testAddMozSearchProvider.js* failed and a workaround was found. A rerun of that test using the failing configuration returns a pass. The failure cannot be replicated reliably, and the workaround is a false positive. These types of failures and workarounds represent one of the largest problems to overcome, as a false positive can add features to the guard that are not protecting against failures.

You can see another example of this in simulation run five shown in table 5.5. The Mozmill test *testSSLDisabledErrorPage.js* results in a failure the first time it is run, however the response given by the Framework is that a workaround has already been

Tests	Simulation 1	Simulation 2	Simulation 3	Simulation 4	Simulation 5	Long Simulation
testPlockupsBlocked.js	W	-	-	-	-	-
testPopupsAllowed.js	-	W	W	-	-	W
testClearFormHistory.js	W	-	-	W	-	W
testSSLDisabledErrorPage.js	-	-	-	-	A(FP)	-
testAddMozSearchProvider.js	W	-	-	-	-	A(FP)
testOpenSearchAutodiscovery.js	-	-	-	W	W	-
testRestoreHomepageToDefault.js	-	-	-	-	-	W(FP)

Table 5.6: Unexpected Workarounds Discovered

found. The test was run by us using the given failing configuration and it passed. This shows the the failure was not replicable, however it did fail during the system run. In addition it must have failed during the replicable check done by the master before. Once the test is determined to replicable the master will check to see if any previously discovered workarounds will suffice. In this case the test passed, but not because of the workaround. The framework will inform the client that a workaround has already been discovered for that failure and that it can continue on.

The results of the unexpected workarounds where a workaround was found are shown in table 5.6. For the five simulation runs there were six Mozmill tests with unexpected workarounds. For these six tests 5 workarounds were discovered. Looking at these workarounds 2 were not true workarounds. Of these one was a workaround that had already been found. This leaves us with one false positive workaround. This workaround results in additional guards for good configurations that are not necessary. There is also a case where an unexpected workaround serves as a workaround for multiple tests. Both test *testPlockupsBlocked.js* and *testPlockupsAllowed.js* fail when *javascript.enabled* is set to false. Because of this once one has failed and has a workaround discovered the other will never fail. *testAddMozSearchProvider.js* and *testOpenSearchAutodiscovery.js* also share a workaround.

5.2.2.3 Guard Activation

In each simulation run we see that the guards will appear after a workaround has been found. This is expected as a failure cannot be guarded against until the workaround is discovered. There are some cases where a test will fail multiple times, and this is attributed to different clients encountering the same failure. In this situation the client who encountered a failure first will receive that a workaround was found and the second client will receive that a workaround already exists. After the workaround has been found no additional clients will encounter the error.

An activated guard will be tied to the failing test case that led to its discovery. Because of this the guard may be preventing additional failures, but only gets credit for stopping a client from avoiding the initial failure. For example in the case of bug 442970 the guard is also avoiding seeing the failure in bug 505548 as the reconfiguration being blocked is necessary to encounter that bug. This means that every time the simulation shows that the guard was being activated for bug 442970, it is also preventing bug 505548.

For each of the simulations once a workaround is found the test case will no longer fail. There are cases where some guards are used more frequently then others, however because seeing the guards requires a random reconfiguration this can be attributed to randomness. The only case where the guard is causing a problem is in the case of false positives. If the workaround given for a false positive is one that is already discovered then there is no issue as no new guard information is discovered. However in the case where a new false positive workaround is discovered the guard will now be blocking a safe reconfiguration.

Test	workaround Found	workaround Not Found	workaround Already Found	notReplicated	Overall
SystemRun1	7.48	N/A	N/A	2.00	4.74
SystemRun2	7.95	N/A	N/A	1.60	4.78
SystemRun3	6.13	11.88	4.65	1.58	6.06
SystemRun4	8.43	7.32	5.18	1.17	5.53
SystemRun5	7.22	N/A	6.03	1.73	4.99
Average	7.43	9.60	5.28	1.62	5.98
LongSystemRun6	9.18	16.22	9.93	1.75	9.27

Table 5.7: Average Time from Failure to Response (min)

5.2.3 Response Time

The response time for each simulation was measured as well and is shown in table 5.7. The response time is the time it takes once a client reports a failure for the master to send a response back. There are four types of responses checked. *workaround found* when a workaround is discovered, *workaround not found* for when no workaround is discovered, *workaround already found* for when an already found workaround will avoid the failure and *not replicated* for when the test passes run under the same configuration it failed on earlier.

The results from the response time table show that the average time to learn that a workaround has been found is almost seven and a half minutes. This is less then the average time to learn that a workaround is not found, at just over nine and a half minutes. For a *workaround not found* response, the algorithm must check all ten iterations of the random algorithm. Because the *workaround found* response time is two minutes less then the workaround not found time we can conclude that the additional time cost by using the minimizer is more then made up by stopping at the first workaround found.

Being stuck in a queue will also increase the time to get a response. If a client is stuck in a queue waiting for a response, and the algorithm is currently trying to find the workaround for a similar bug, it is possible that the workaround can be reused

without rerunning the test. This means that even though a client was stuck in the queue they can receive a faster response then the client that was in front of them. This is shown by the fact that the *workaround already found* average response time is two minutes less then the workaround found response at only five and a half minutes.

The first thing the failure avoidance algorithm tries to do before anything else is replicate the failure. If the failure cannot be effectively replicated then a workaround cannot be discovered. The algorithm takes the configuration from the failing test case, and reruns the test checking the result. If it passes then the test is considered not replicable and a *not replicated* result is returned to the client so they can move on. An average time for response of less then two minutes was reported.

Overall, for five runs of the simulation over the course of three hours, each the average time for a response was less then six minutes. We compare this to the results of the long simulation run (described next) and it shows an increase in time for response. The overall time to a response for the long system run takes over nine minutes. This is an increase in about 60% from the earlier response. This is mostly due to an increase in the response time for workarounds not found, which makes sense as there were a lot more workaround not found results in the long simulation which increases the odds that a queue will form and the response for the last client in the queue will be workaround not found.

These response times are heavily tied to the queue, which means that as the number of clients increase, the odds of a longer queue and longer wait times increase as well. Improving the response time can be done in two ways. The first is to get a faster response by using an algorithm that returns a workaround found or workaround not found response quicker. The second is to add additional failure avoidance algorithms that can run in parallel. This would allow the failure avoidance framework to handle multiple simultaneous failures encountered by the clients.

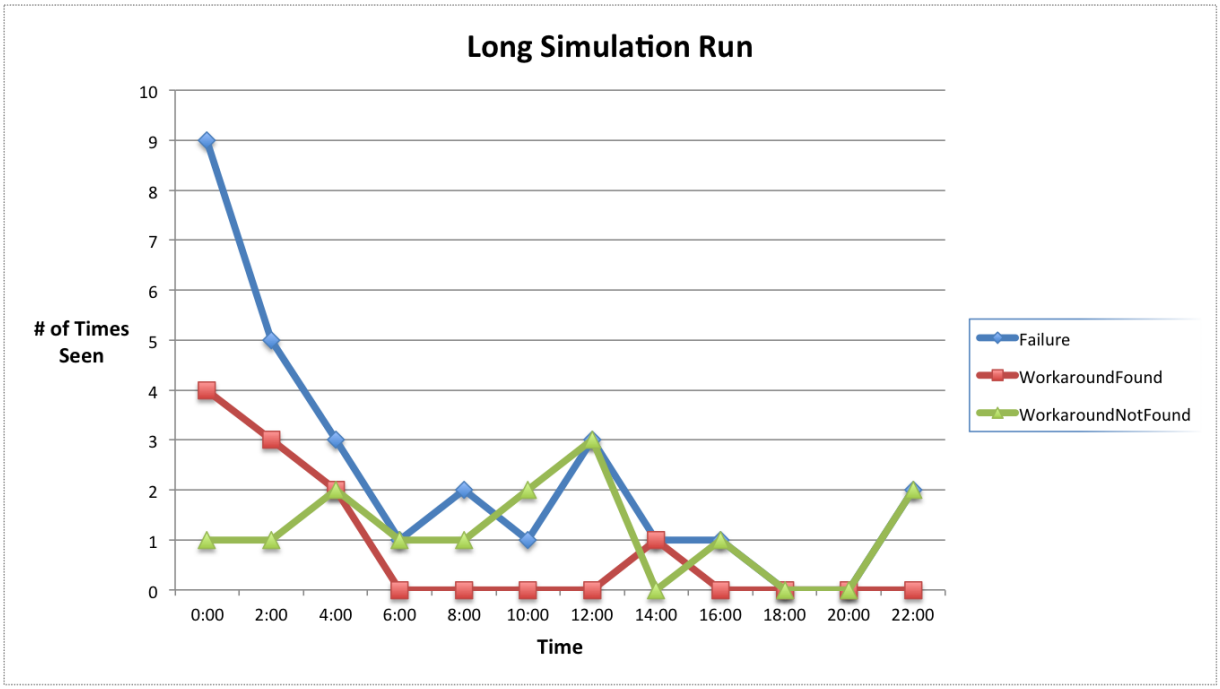

Figure 5.1: Simulation of Long System Run Graph

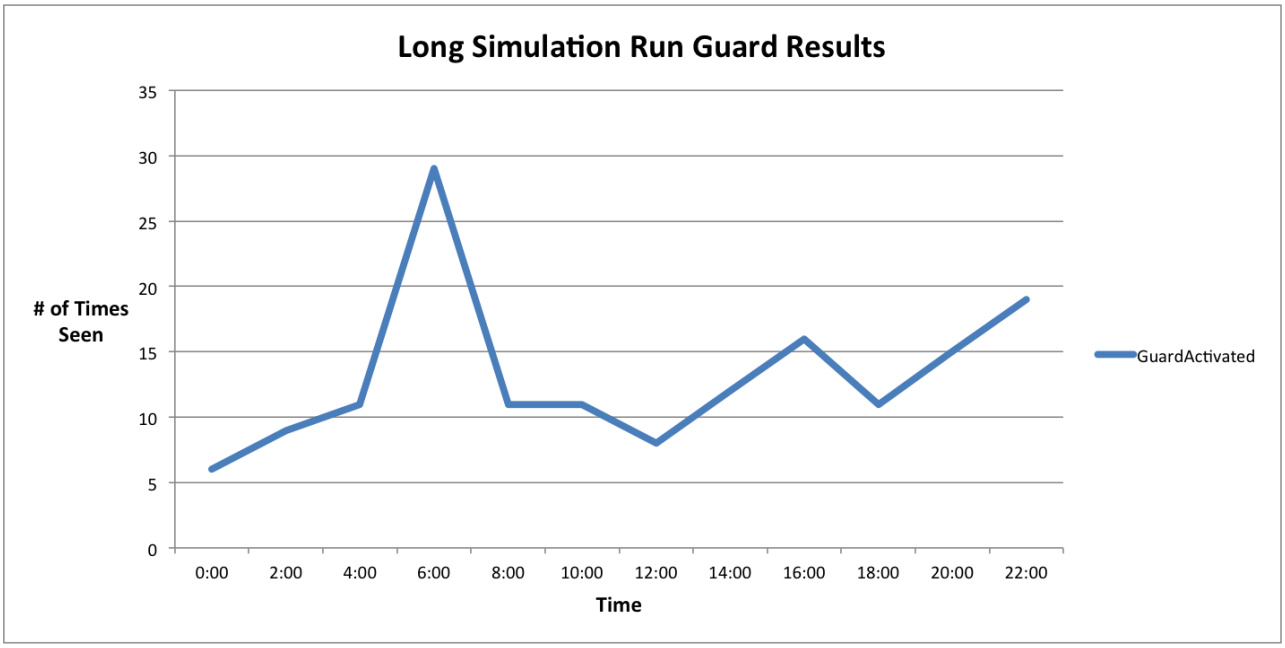

Figure 5.2: Simulation of Long System Run Guard Activation Graph

Test	2:00	4:00	6:00	8:00	10:00	12:00	14:00	16:00	18:00	20:00	22:00	24:00
No workaorund Found												
testGoButton.js	–	–	–	–	–	–	–	–	–	–	–	FN
testCloseDownloadManager.js	–	–	FN	–	–	–	FN	–	–	–	–	FN
testAutoCompleteOff.js	–	–	FN	–	–	–	–	–	–	–	–	-
testDisableFormManager.js	–	–	–	–	F	NFN	FN	–	FN	–	–	-
testOpenSearchAutodiscovery.js	–	–	–	FN	–	–	–	–	–	–	–	–
testDefaultSecurityPrefs.js	FN	–	–	–	–	–	–	–	–	–	–	–
testSearchViaFocus.js	–	–	–	–	FN	–	–	–	–	–	–	-
testSSLDisabledErrorPage.js	–	FN	–	–	–	–	–	–	–	–	–	–
testHomeButton.js	–	–	–	–	–	–	FN	–	–	–	–	-
Workaorund Found												
firefoxBug_344189.js	2FWAG	–	G	2G	G	G	–	–	2G	–	G	2G
firefoxBug_306208.js	2FWAG	3G	–	5G	3G	–	G	4G	2G	–	3G	2G
firefoxBug_797945.js	2FWAG	2G	–	G	–	G	G	–	G	G	3G	-
firefoxBug_808290.js	–	–	FWG	5G	2G	G	2G	–	3G	2G	–	2G
firefoxBug_840411.js	–	FW	4G	G	–	–	–	–	–	–	G	-
firefoxBug_442970.js	F	WG	G	4G	G	2G	2G	G	G	G	G	G
testClearFormHistory.js	FWG	3G	–	5G	G	G	G	3G	3G	2G	G	3G
testPopupsAllowed.js	–	F	W2G	2G	G	4G	G	G	3G	G	2G	4G
testAddMozSearchProvider.js	2G	FWF	A2G	4G	2G	G	–	3G	–	3G	3G	-
testRestoreHomepageToDefault.js	–	–	–	–	–	–	–	FW	G	G	–	5G

Table 5.8: Simulation of Long System Run

Result	0:00	2:00	4:00	6:00	8:00	10:00	12:00	14:00	16:00	18:00	20:00	22:00
Failure	9	5	3	1	2	1	3	1	1	0	0	2
WorkaroundFound	4	3	2	0	0	0	0	1	0	0	0	0
WorkaroundNotFound	1	1	2	1	1	2	3	0	1	0	0	2
GuardActivated	6	9	11	29	11	11	8	12	16	11	15	19

Table 5.9: Simulation of Long System run Overall

5.2.4 Long Simulation Overall Results

We also ran a longer simulation. The long simulation ran for 24 hours. The tests that encountered a failure are displayed in Table 5.9. During this experiment, six of the seven Firefox bugs had workarounds discovered. The one that did not have a workaround discovered was bug 505548 which makes sense as that bug was being guarded by the workaround found for bug 442970. This shows that the failure avoidance framework is working as intended and it shows that given enough time all of the expected workarounds will be discovered. We also see that the guard is working and prevents any of these failures from reoccurring once the workaround has been found.

In addition there were workarounds discovered for an additional four failing tests.

Two of these were true workarounds, while the other two are false positives. There were also 14 no workaround found responses for eight Mozmill tests. The increase in failures might be due to the configuration that the clients are using. However, if the configuration is responsible we would expect the failure avoidance algorithm to find a workaround. The other possibility is that as the test goes on there is a larger chance for timing issues or file corruption within sandhills.

Looking at a graph of the overall results shown in Figure 5.1, we see how the number of failures decreases as time goes on, eventually hitting a point where it overlaps with the no workaround found line. This represents a point where each failure does not produce a workaround. The graph has three lines. One for number of failures, one for workarounds found and one for workarounds not found. This data shows that during a run of the simulation, all of the known failures are avoided within the first eight hours, and only one additional workaround is discovered after another 16 hours. There are however, a number of failures where no workaround is found during the entire simulation. Understanding and reducing these failures is a large part of future work.

The number of guards activated during the long simulation are shown in Figure 5.2 Looking at the rate of which the guard is activated it seems that the number increases as the test goes on, and then jumps to its highest point once the majority of workarounds are found. This makes sense as a big part of hitting a guard is luck when dealing with random reconfigurations. A larger number of workarounds found means a greater chance of hitting a bad configuration and forcing a guard to activate.

Test	2:00	4:00	6:00	8:00	10:00	12:00	14:00	16:00	18:00	20:00	22:00	24:00
No workaorund Found												
testGoButton.js	–	–	–	–	–	–	–	–	–	–	–	FN
testCloseDownloadManager.js	–	–	FN	–	–	–	–	–	–	–	–	–
testAutoCompleteOff.js	–	–	FN	–	–	–	–	–	–	–	–	-
testDisableFormManager.js	–	–	–	–	F	N	–	–	–	–	–	-
testOpenSearchAutodiscovery.js	–	–	–	FN	–	–	–	–	–	–	–	–
testDefaultSecurityPrefs.js	FN	–	–	–	–	–	–	–	–	–	–	–
testSearchViaFocus.js	–	–	–	–	FN	–	–	–	–	–	–	-
testSSLDisabledErrorPage.js	–	FN	–	–	–	–	–	–	–	–	–	–
testHomeButton.js	–	–	–	–	–	–	FN	–	–	–	–	-
Workaorund Found												
firefoxBug_344189.js	2FWAG	–	G	2G	G	G	–	–	2G	–	G	2G
firefoxBug_306208.js	2FWAG	3G	–	5G	3G	–	G	4G	2G	–	3G	2G
firefoxBug_797945.js	2FWAG	2G	–	G	–	G	G	–	G	G	3G	-
firefoxBug_808290.js	–	–	FWG	5G	2G	G	2G	–	3G	2G	–	2G
firefoxBug_840411.js	–	FW	4G	G	–	–	–	–	–	–	G	-
firefoxBug_442970.js	F	WG	G	4G	G	2G	2G	G	G	G	G	G
testClearFormHistory.js	FWG	3G	–	5G	G	G	G	3G	3G	2G	G	3G
testPopupsAllowed.js	–	F	W2G	2G	G	4G	G	G	3G	G	2G	4G
testAddMozSearchProvider.js	2G	FWF	A2G	4G	2G	G	–	3G	–	3G	3G	-
testRestoreHomepageToDefault.js	–	–	–	–	–	–	–	FW	G	G	–	5G

Table 5.10: Simulation of Long System Run with Unique Failures

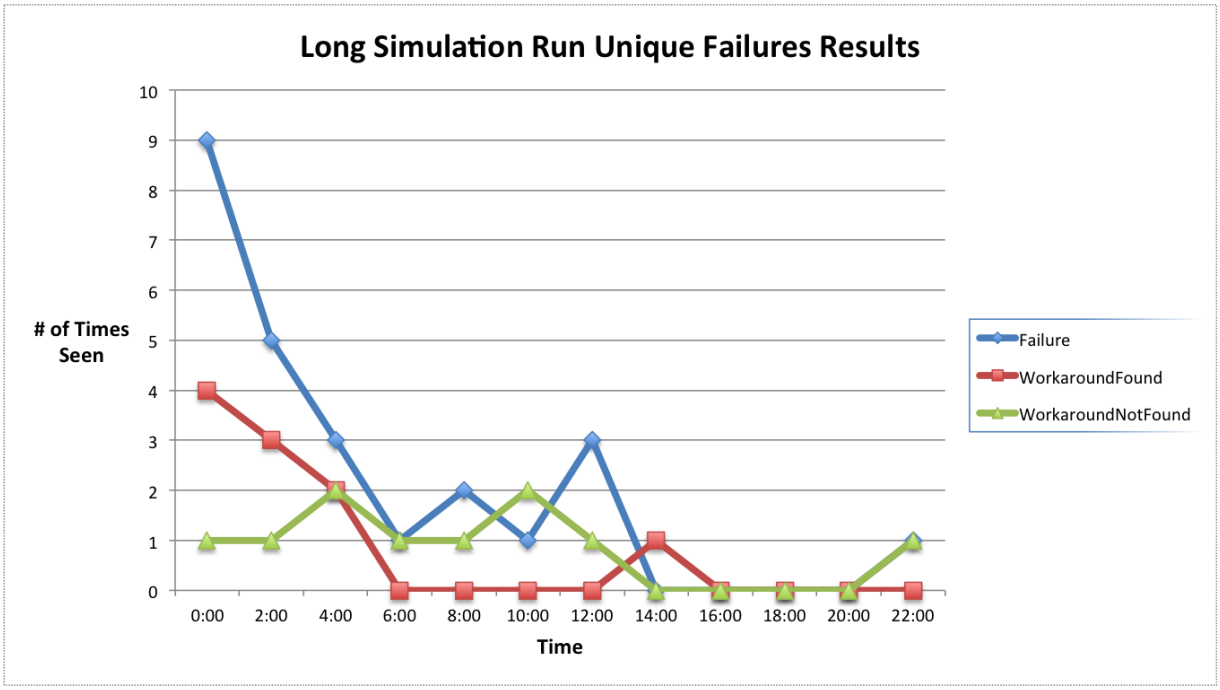

Figure 5.3: Simulation of Long System Run with Unique Failures Graph

5.2.4.1 Long Simulation Overall Results for Unique Failures

Looking at the failures in 5.9 there are some failures that occur multiple times for the same test. It can be assumed that these are really the same failure, being encountered again, rather then a new failure. If this is true then the graph will look slightly differently because we can remove the duplicate failures. Removing these duplicate failures results in table 5.10.

The new graph shown in figure 5.3 has a more obvious drop in failures, and the number of failures found after 14 hours of run time is only one. This graph shows how after a period of time the simulation will reach a steady state, where very few failures are encountered, and those that are have no workaround. In the case of the long simulation the steady state is reached after 14 hours of runtime.

5.3 Discussion of Results

The number of unexpected workarounds was surprising. The unexpected workarounds that are not false positives show us that the failure avoidance framework does work, since these faults are native to the system and not seeded (as our original seven failing test cases are).

The choice of the random algorithm, 10 iterations, was made to keep the cost of failure avoidance low. However, we believe that the covering array has the potential perform better based on the results from the GCC study. We plan to investigate this as future work. The framework is designed to be interchangeable with regards to how the workarounds were found, which means that adapting the covering array for the Firefox simulation is easy. For future work this would seem to be the better option, as any cases of a workaround not being found using the covering array will always return workaround not found. Using the random it is possible that a series of unlucky

Tests	Simulation 1	Simulation 2	Simulation 3	Simulation 4	Simulation 5	Long Simulation
firefoxBug_344189.js	W	W	W	W	W	W
firefoxBug_306208.js	-	-	-	W	W	W
firefoxBug_797945.js	W	W	W	W	W	W
firefoxBug_808290.js	-	-	W	-	W	W
firefoxBug_840411.js	-	W	W	-	W	W
firefoxBug_442970.js	-	-	W	-	-	W
firefoxBug_505548.js	-	-	-	-	-	-

Table 5.11: Expected Workarounds Discovered

configuration choices will prevent a workaround from being found.

5.4 Summary

For research question one we asked if the failure avoidance framework can find all of the expected workarounds. The results for expected workarounds found are shown in Table 5.11. Over the course of the five three hour simulations six of the seven expected workarounds are found. Simulation three and simulation five find the most workarounds at five each. Simulation four and simulation two each find three, while simulation one only finds two expected workarounds. This can be attributed to the random nature of the simulation. Finding the expected workarounds requires a client to be in a dangerous state and run the appropriate test. Given more time, the odds of this happening increases. The long simulation finds six of the seven expected workarounds within the first six hours.

We can answer question one in the affirmative, as even though no workaround was found for bug 505548 this can be attributed to the fact that once a workaround was found for bug 442970 then bug 505548 will never fail. Even in the relatively short runtime of three hours the bugs that fail on the default configuration will have a workaround discovered, and at best all but one of the bugs will have a workaround discovered. Increasing the time shows that all of the expected workarounds can be discovered during a single simulation with four clients.

For research question two we asked if the guard is effective at preventing future workarounds. The guards are activated in each simulation once a workaround has been discovered. Over the course of the five simulations in Table 5.1−5.5 there are 24 cases where a workaround is discovered for some failure. In all 24 cases the test does not fail again during the simulation. The long simulation in table 5.9 adds an additional 10 workarounds discovered, and all 10 of those tests do not fail again once the workaround is discovered.

The guard is preventing the clients from running a test while in a potentially unsafe configuration. The tests are run and passed after the workaround has been discovered, so it is not as if the failure is not encountered simply because the test isn't being run. There is also no case where a failing test is seen three times before a workaround is discovered. With four clients the maximum number of times a failure is seen before a workaround is found is two, which means that during the course of all simulations no more then two clients are ever inconvenienced by a single failing test case.

We can answer question two in the affirmative, as once a workaround is discovered for some test there is no case where the test fails again. The guard has a 100% success rate during the simulation of preventing seeing the same failure again. There is also the fact that a guard can be preventing additional failures from occurring, such as in the case with bug 442970 and bug 505548 and in the case with *testPopupsBlocked.js* and *testPopupsAllowed.js*. Finding a workaround for one of the test pairs will serve as a workaround for the other, and the guard is effectively preventing seeing a previously discovered failure and one that has not yet been seen.

Chapter 6

Conclusions and Future Work

In this thesis we implemented an extension to Rainbow which allows the self adaptive framework to avoid failures by automatically finding workarounds in the configuration space. We call this the Failure Avoidance Framework. Finding the workarounds is done by using a modified version of the failure avoidance algorithm, which is no longer bound by a one or two-hop algorithm.

In extending the failure avoidance algorithm we discovered additional workarounds for gcc bugs beyond the two-hops that had previously been found. These workarounds go up to five hops away from the starting configuration. There are also bugs that did not have a one or two-hop workaround, but do have a three-hop workaround. Three new techniques were tried. A genetic algorithm, random with minimizer, and covering Array with minimizer. From these techniques, the genetic algorithm was shown to be not as good at finding multiple algorithms, and the problem does not have a simple fitness function. The random technique performed well, but by nature cannot guarantee the same results every time, which leads to the possibility of missing workarounds. The covering array is hampered by configuration masking at lower strengths, but this can be mitigated by increasing the number of covering arrays or

improving the strength.

The failure avoidance algorithm was implemented on sandhills and a number of simulations were run to prove that the system works. A test suite was built using Mozmill tests and a seven configuration bugs from the Firefox bug repository. Four firefox clients were implemented which randomly change their configuration and then randomly run a Mozmill tests during a simulation. If a test fails a probe records the failure, a gauge changes the property in the failure avoidance framework model which violates a constraint, this activates the adaptation manager which selects the find workaround strategy. This strategy has a tactic which when executed calls an effector and the effector runs a failure avoidance algorithm and updates the client with the result. The process is repeated, and eventually all expected failures will be encountered, and workarounds will be discovered which stops the failure from ever being encountered again.

Beyond this the failure avoidance framework will also detect unexpected workarounds. Four of these workarounds are false positives and the result is that the clients will be guarding against safe configurations. Three of them however are not false positives and are actual configuration based failures in the standard Mozmill test suite. This shows that the failure avoidance framework can avoid failures beyond those we placed in the simulation.

6.1 Future Work

With respect to the failure avoidance algorithm we wanted to take a look at additional techniques. There has been a lot of work done on adapting covering arrays to large configurable system. One of these methods is to introduce prioritization into the algorithm that generates the covering array. This would steer the covering array to

focus on particular feature options that are more likely to be involved in a workaround. Using an understand of the target system we would be able to tailer the failure avoidance algorithm and hopefully further improve its effectiveness and efficiency.

There is a large potential for future work in the area of automatic failure avoidance. The simulation was run using the failure avoidance algorithm with random and the minimizer, however the covering array performed very well on the gcc tests. We want to see how well the covering array would work on the larger configuration space of Firefox, and the results it has on the response time and number of workarounds found. Particularly with the unexpected workarounds and the workaround not found responses from the long simulation.

Another aspect of the failure avoidance framework is how multiple failure avoidance algorithms could be adapted. The framework is currently limited by a single failure avoidance algorithm. Further work could improve this system, so that depending on the type of error, or the client sending it, different failure avoidance algorithms could be used. If the client wants an immediate response a very fast failure avoidance algorithm is used. If only one client has encountered an failure and there is no queue then a more effective algorithm could be used to find additional workarounds that might help prevent future failures.

One of the problems we encountered during the simulation were the false positive workarounds. Understanding what causes the non replicable failure and reducing these cases is a big part of future work. There is also the issue of failures where no workaround is found. More research needs to be done on these failures to discover what is the cause of them. If a change in configuration is responsible for the failure, then it would make sense that the failure avoidance algorithm should find a workaround. This however is not the case and understanding why is big question.

Finally, we plan to continue to generalize the failure avoidance framework. RIght

now it is still partially tied to the chosen Firefox target system. Even though the target system can be swapped out, some of the existing code was specifically tailored to integrate to Firefox. By using more general code we can make it even easier for the failure avoidance framework to be adapted for a large number of different systems.

Appendix A

Sandhills

Sandhills is a linux cluster that was used to run all of the experiments in this thesis. Using sandhills we can deploy multiple jobs in parallel, which allows us to minimize the time spent running tests. For the GCC tests the bugs were split into groups, and each group could be run simultaneously with the results combined at the end. For the Firefox tests each of the delegates and the master were deployed to their own node. This allows us to simulate a system with 4 users and a master, while only using one command on a single machine.

```
#!/bin/bash
#SBATCH --job-name=GCCBFMM
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=5
#SBATCH --time=100:00:00
#SBATCH
    --output=/work/esquared/jswanson/gccBFMM/gcc/results/isolation/BFMM_Results.STDOUT
#SBATCH
    --error=/work/esquared/jswanson/gccBFMM/gcc/results/isolation/BFMM_Results.OUT
#SBATCH --mem-per-cpu=8172
#SBATCH --partition=esquared

srun ./startLocp.sh
```

Figure A.1: Example Code to Launch a Job on Sandhills

```
#!/bin/bash
STNAME=‘hostname‘
IP=‘dig +short $HOSTNAME‘
echo $SLURM_LOCALID

if [ "$SLURM_NODEID" -eq "0" ] ; then
for i in {0..4}
do
if [ "$SLURM_LOCALID" -eq "$i" ] ; then
       NUM=$[$i+1]
        { time ./mainDeployBFMM_4_4_0.sh $NUM ; } 2> results/isolation/4_4_0
fi
done
fi
if [ "$SLURM_NODEID" -eq "1" ] ; then
for i in {0..4}
do
if [ "$SLURM_LOCALID" -eq "$i" ] ; then
        NUM=$[$i+1]
        { time ./mainDeployBFMM_4_4_1.sh $NUM ; } 2> results/isolation/4_4_1
fi
done
fi
if [ "$SLURM_NODEID" -eq "2" ] ; then
for i in {0..4}
do
if [ "$SLURM_LOCALID" -eq "$i" ] ; then
        NUM=$[$i+1]
        { time ./mainDeployBFMM_4_4_2.sh $NUM ; } 2> results/isolation/4_4_2
fi
done
fi
```

Figure A.2: Example Code to Run Scripts in Parallel on Sandhills

Appendix B

Mozmill

Mozmill is an add-on for firefox that automates the testing process. It uses test cases written in javascript. It recreates the actions of a user and can check for events to declare if the test passed or failed. There exists a test suite for Mozmill that can be automated which makes it perfect for testing changes made to Firefox. If a Mozmill test fails, then it can be said that some functionality of Firefox has been changed from the expected. The mozmill tests use an assert or expected function to check for functionality.

```
/* This Source Code Form is subject to the terms of the Mozilla Public
 * License, v. 2.0. If a copy of the MPL was not distributed with this
 * file, You can obtain one at http://mozilla.org/MPL/2.0/. */

// Include required modules
var { expect } = require("../../../lib/assertions");
var places = require("../../../lib/places");
var toolbars = require("../../../lib/toolbars");
var utils = require("../../../lib/utils");


const TIMEOUT = 5000;

const LOCAL_TEST_FOLDER = collector.addHttpResource('../../../data/');
const LOCAL_TEST_PAGE = LOCAL_TEST_FOLDER + 'layout/mozilla_contribute.html';

var setupModule = function() {
  controller = mozmill.getBrowserController();
  locationBar = new toolbars.locationBar(controller);
}

var teardownModule = function() {
  places.restoreDefaultBookmarks();
}

var testFailure344189 = function() {
       const Cu = Components.utils;

       Cu.import("resource://gre/modules/Services.jsm");

       const PREF_SEARCH_OPENINTAB = "browser.search.openintab";

       var result = Services.prefs.getBoolPref(PREF_SEARCH_OPENINTAB);

       expect.ok(!result, "browser.search.openintab set to false");

}
```

Figure B.1: Example Mozmill Test

```
../Python-Install/bin/python2.7 ../Python-Install/bin/mozmill -b
    ../obj-x86_64-unknown-linux-gnu/dist/bin/firefox -p
    /work/esquared/jswanson/rainbow/FirefoxClientSystem1/profile -P 6711 -t
    /work/esquared/jswanson/rainbow/FirefoxClientSystem1/runMozmillTests/mozmill-tests/tests/func
    >> testResults.txt
```

Figure B.2: Example Code to Run a Mozmill Tests

Appendix C

Example Logs

```
1389319001 test passed : /testFailures/firefoxBug_840411.js
1389319018 test failed : /testFailures/firefoxBug_797945.js
1389319301 Updating guard
1389319302 Workaround found for /testFailures/firefoxBug_797945.js
1389319320 test passed : /testDownloading/testOpenDownloadManager.js
1389319343 test passed : /testSearch/testSearchViaFocus.js
1389319361 test passed : /testSearch/testOpenSearchAutodiscovery.js
1389319378 test passed : /testToolbar/testBackForwardButtons.js
1389319397 test passed : /testPreferences/testPaneRetention.js
1389319415 test passed : /testTabView/testToggleTabView.js
1389319433 test passed : /testTabbedBrowsing/testCloseTab.js
1389319452 test passed : /testSecurity/testDVCertificate.js
1389319469 test failed : /testFailures/firefoxBug_306208.js
1389319590 Updating guard
1389320066 Updating guard
1389320068 Workaround found for /testFailures/firefoxBug_306208.js
1389320090 test passed : /testToolbar/testHomeButton.js
1389320110 test passed : /testSecurity/testSSLDisabledErrorPage.js
1389320127 test passed : /testToolbar/testBackForwardButtons.js
1389320151 test failed on teardown (skip): /testSearch/testSearchViaFocus.js
1389320173 test passed : /testSecurity/testIdentityPopupOpenClose.js
1389320175 GUARD ACTIVATED user_pref("plugins.click_to_play", false); has been
    linked to potential errors
1389320190 test failed : /testFormManager/testClearFormHistory.js
1389320528 Error could not be replicated /testFormManager/testClearFormHistory.js
1389320547 test passed : /testDownloading/testOpenDownloadManager.js
1389320564 test passed : /testSecurity/testIdentityPopupOpenClose.js
1389320582 test passed : /testFailures/firefoxBug_808290.js
1389320600 test passed : /testPreferences/testSwitchPanes.js
1389320618 test passed : /testAwesomeBar/testPasteLocationBar.js
1389320641 test passed : /testPreferences/testRestoreHomepageToDefault.js
1389320659 test passed : /testTabView/testToggleTabView.js
1389320678 test passed : /testTabView/testTabGroupNaming.js
1389320697 test passed : /testLayout/testNavigateFTP.js
1389320725 test passed : /testFormManager/testDisableFormManager.js
1389320742 test passed : /testFailures/firefoxBug_344189.js
```

Figure C.1: Example Client Log

Bibliography

[1] *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] Free Software Foundation. gnu 4.1.1 manpages., 2005.

[3] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 782–791, Piscataway, NJ, USA, 2013. IEEE Press.

[4] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 237–246, New York, NY, USA, 2010. ACM.

[5] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Self-healing by means of automatic workarounds. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 17–24, New York, NY, USA, 2008. ACM.

[6] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic,

Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[7] Shang-Wen Cheng. *Rainbow: Cost-effective, Software Architecture-based Self-adaptation*. PhD thesis, Carnegie Mellon University, 2008.

[8] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860 – 2875, 2012.

[9] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Making self-adaptation an engineering reality. In Ozlap Babaoghu, Mark Jelasity, Alberto Montroser, Christof Fetzer, Stefano Leonardi, and Aad Van Moorsel, editors, *Proceedings of the Conference on Self-Star Properties in Complex Information Systems*, volume 3460 of *LNCS*. Springer-Verlag, 2005. Also available from Springer-Verlag here.

[10] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Shanghai, China, 21-22 May 2006.

[11] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches*

to Software Engineering, FASE'08/ETAPS'08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[13] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[14] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engingeering*, 34(5):633–650, 2008.

[15] Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Sample spaces and feature models: There and back again. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 22–31, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM.

[17] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Test-and-adapt: An approach for improving service interchangeability. *ACM Trans. Softw. Eng. Methodol.*, 22(4):28:1–28:43, October 2013.

[18] Nicholas DiGiuseppe and James A. Jones. Fault interaction and its repercussions. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

[19] Ali Ebnenasir. Designing run-time fault-tolerance using dynamic updates. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '07, pages 15–, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[21] Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 234–244, New York, NY, USA, 2011. ACM.

[22] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, January 2003.

[23] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[24] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. *Increasing System Dependability through Architecture-based Self-repair*, pages 61–89. Springer-Verlag, 2003.

[25] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.

[26] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering*, SSBSE '09, pages 13–22, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems*, ASAS '11, pages 24–33, New York, NY, USA, 2011. ACM.

[28] BradyJ. Garvin, MyraB. Cohen, and MatthewB. Dwyer. Failure avoidance in configurable systems through feature locality. In Javier Cmara, Rogrio Lemos, Carlo Ghezzi, and Antnia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 266–296. 2013.

[29] John C. Georgas, André van der Hoek, and Richard N. Taylor. Architectural runtime configuration management in support of dependable self-adaptive software. In *Proceedings of the 2005 Workshop on Architecting Dependable Systems*, WADS '05, pages 1–6, New York, NY, USA, 2005. ACM.

[30] Hassan Gomaa and Mohamed Hussein. Model-based software design and adaptation. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '07, pages 7–, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Alessandra Gorla. Automatic workarounds as failure recoveries. In *Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium*, FSEDS '08, pages 9–12, New York, NY, USA, 2008. ACM.

[32] Alan Hartman. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms*, pages 237–266. Springer, 2005.

[33] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[34] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, June 2004.

[35] Jie Li, Changhai Nie, and Yu Lei. Improved delta debugging based on combinatorial testing. In *Proceedings of the 2012 12th International Conference on Quality Software*, QSIC '12, pages 102–105, Washington, DC, USA, 2012. IEEE Computer Society.

[36] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Control*, 20(3-4):567–604, September 2012.

[37] Mozilla. Mozmill bug repository.

[38] Mozilla. Firefox version 18.0, 2012.

[39] Mozilla. Mozilla bug repository, 2013.

[40] Changhai Nie and Hareton Leung. The minimal failure-causing schema of combinatorial testing. *ACM Trans. Softw. Eng. Methodol.*, 20(4):15:1–15:38, September 2011.

[41] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011.

[42] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[43] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Control*, 20(3-4):605–643, September 2012.

[44] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis*, pages 75–85, July 2008.

[45] Xiao Qu. *Configuration Aware Prioritization for Regression Testing*. PhD thesis, Lincoln, NB, USA, 2010. AAI3412925.

[46] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 75–86, New York, NY, USA, 2008. ACM.

[47] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 445–454, New York, NY, USA, 2010. ACM.

[48] Brian Robinson and Lee White. Testing of user-configurable software systems using firewalls. In *International Symposium on Software Reliability Engineering*, pages 177–186, November 2008.

[49] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[50] Norbert Siegmund, Mario Pukall, Michael Soffner, Veit Köppen, and Gunter Saake. Using software product lines for runtime interoperability. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, pages 4:1–4:7, New York, NY, USA, 2009. ACM.

[51] Hema Srikanth, Myra B. Cohen, and Xiao Qu. Reducing field failures in system configurable software: Cost-based prioritization. In *Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, ISSRE'09, pages 61–70, Piscataway, NJ, USA, 2009. IEEE Press.

[52] Elisabeth A. Strunk and John C. Knight. Assured reconfiguration of embedded real-time software. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 367–, Washington, DC, USA, 2004. IEEE Computer Society.

[53] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International*

Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, pages 202–207, New York, NY, USA, 2011. ACM.

[54] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[55] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan 2006.

[56] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 45–54, New York, NY, USA, 2004. ACM.

[57] Pamela Zave. Feature interactions and formal specifications in telecommunications. *Computer*, 26(8):20–29, August 1993.

[58] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

[59] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.