Beyond the Rainbow: Self-Adaptive Failure Avoidance in Configurable Systems

Jacob Swanson, Myra B. Cohen, Matthew B. Dwyer, Brady J. Garvin and Justin Firestone
Dept. of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115, USA
{jswanson,myra,dwyer,bgarvin,jfiresto}@cse.unl.edu

ABSTRACT

Self-adaptive software systems monitor their state and then adapt when certain conditions are met, guided by a global utility function. In prior work we developed algorithms and conducted a post-hoc analysis demonstrating the possibility of adapting to software failures by judiciously changing configurations. In this paper we present the REFRACT framework that realizes this idea in practice by building on the self-adaptive Rainbow architecture. REFRACT extends Rainbow with new components and algorithms targeting failure avoidance. We use REFRACT in a case study running four independently executing Firefox clients with 36 passing test cases and 7 seeded faults. The study show that workarounds for all but one of the seeded faults are found and the one that is not found never fails – it is guarded from failing by a related workaround. Moreover, REFRACT finds workarounds for eight configuration-related unseeded failures from tests that were expected to pass (and did under the default configuration). Finally, the data show that when a failure and its workaround are found, configuration guards prevent the failure from appearing again. In a simulation lasting 24 hours we see over 150 guard activations and no failures with workarounds remaining beyond 16 hours.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Experimentation

Keywords

Self-Adaptive Software, Configurable Software

1. INTRODUCTION

Many programs provide users with the option of selecting a custom set of features that can be combined, almost arbitrarily, at runtime. For instance, a web server like Apache provides options to configure, for example, permissions, server-side include support, access logging, host name lookup, buffer sizes, and server capacity. Adding or removing features may improve the robustness of a system, improve performance, or simply improve the user experience. A typical configurable system may have hundreds or thousands of options, and this leads to billions or trillions of possible customizations; Apache has more than 10^{55} possible configurations [11].

Research on testing configurable systems has focused on a particular type of fault called a *feature interaction fault*. Such a fault triggers only under a specific combination of features [11,22,27,37,39,45]. The problem of feature interaction faults has existed for years [46]. These types of faults are difficult to detect, and therefore often hide in released software. Once deployed, they may trigger infrequently making them hard to reliably detect and repair.

Self-adaptive software systems are a new breed of software that provide adaptation mechanisms that allow continued operation when the system environment changes. For example, if an application utilizes too much CPU, it may reconfigure resource settings to achieve a desired maximum utilization. Most work on self-adaptation has focused on continuous quality attributes allowing techniques from feedback-control to be applied [6, 9, 12, 16, 20, 24, 40]. Recent work by Carzaniga et al. [2] uses self-adaptation in a different way – to avoid executing faulty code after a failure is observed. Their code adaptation automates a kind of *workaround* – avoiding rather than repairing the fault by modifying the code to use alternative libraries. This approach suggests that self-adaptation may successfully utilize discrete events to drive adaptation, in addition to using continuous measures of quality attributes.

In prior work we also explored the use of failures to drive adaptation [21]. In our approach, adaptation is performed by reconfiguring the system through its pre-defined configuration options. We developed several algorithms for adaptive reconfiguration and performed a case study using the fault history of the GNU `gcc` compiler – accessed through its bug database. Our findings demonstrate that the algorithms we developed are able to avoid failures in configurable systems through adaptive reconfiguration. In this paper, we take this work further by building a flexible framework for failure avoidance via reconfiguration on top of an existing self-adaptive system. Our framework, REFRACT (REconfiguration-based FailuRe AvoidanCe Technique), extends the Rainbow self-adaptive system [7] with new meth-

FSE'14 , November 16–22, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...$15.00.

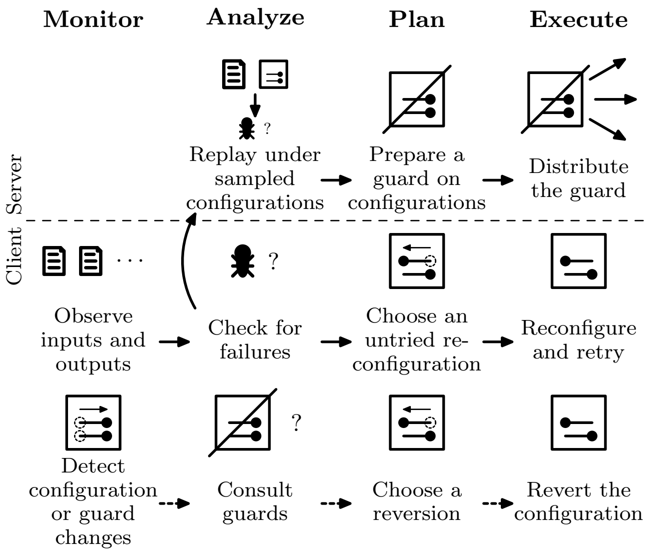
Monitor
Analyze
Plan
Execute
Server
Client
Replay under sampled configurations
Prepare a guard on configurations
Distribute the guard
Observe inputs and outputs
Check for failures
Choose an untried re-configuration
Reconfigure and retry
Detect configuration or guard changes
Consult guards
Choose a reversion
Revert the configuration

Figure 1: REFRACT MAPE Loops

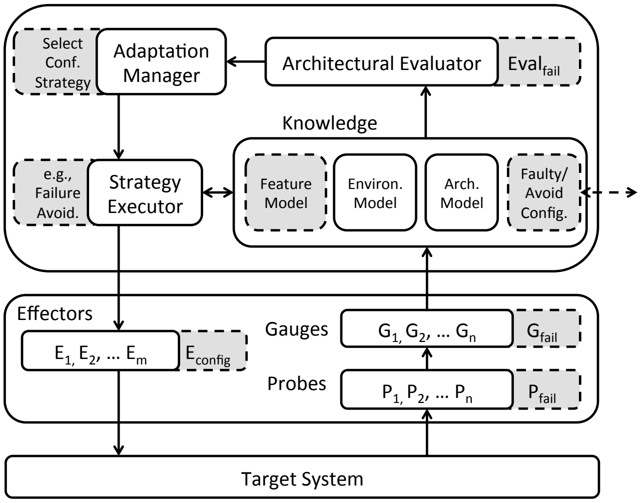

Figure 2: REFRACT Rainbow Extension

ods to monitor for discrete failures, strategies for reconfiguring to avoid failures and to guard against future failures.

REFRACT is able to observe and drive fault-adaptation in a set of simultaneously executing target applications. In this paper, we use REFRACT to avoid failures in independently executing instances of Firefox. REFRACT is instantiated with a model of the Firefox configuration space and failure avoidance strategies, as well as methods that restart the application in a new configuration. In a case study we evaluate the effectiveness of REFRACT on a set of real faults for the Firefox web browser. We show that self-adaptation for avoidance of configuration dependent failures does find and then subsequently avoid failures in the system.

While REFRACT is not yet highly optimized, it demonstrates the feasibility of the approach – failures are diagnosed and Firefox is reconfigured to avoid them in a matter of minutes. Moreover, over time REFRACT can collect information from a set of deployed Firefox instances and distribute that information back to those instances. This allows all of the deployed instances to be protected against future failures when just a single instance encounters a failure. We also see protection derived from one workaround that incidentally protects against a different failure.

The contributions of this paper are: (1) a framework (REFRACT) for self-adaptation to avoid feature-related failures; (2) a prototype instance of REFRACT that supports the Firefox browser; and (3) the results of a case study using REFRACT to avoid failures in Firefox that uses real faults to show the feasibility and potential of the approach.

We describe the main use cases and background for our work in the next section. We follow that with a description of our framework in Section 3. We then present our case study and results in Sections 4 and 5. We discuss related work in Section 6 and conclude and present future work in Section 7.

2. OVERVIEW AND BACKGROUND

Consider a configurable target application, such as Firefox, running under the control of REFRACT, which we will call Firefox-REFRACT. Figure 1 illustrates the three self-adaptive behaviors that REFRACT supports as MAPE (monitor, analyze, plan, and execute) loops [6].

The top two rows, with solid arrows, depict Firefox-REFRACT reacting to a failure, starting with the client-side Firefox encountering the bug. There, REFRACT monitors Firefox by *observing inputs and outputs* during execution and *checking for failures*; we focus just on failures in this explanation for simplicity. When a failure is detected, the recorded inputs and the current configuration are sent to the server, where the problem is analyzed by *replaying it under sampled configurations.* If the failure can be recreated as reported, but the execution passes under some configurations in the sample, the server *prepares a guard on configurations* based on the differences between the passing and failing configurations. Finally, it *distributes the guard* to **all** running Firefox-REFRACT clients so that they can use it to avoid both the original and similar failures. Because of the way the guard is encoded, it can also suggest reconfigurations that will take a client out of the situations it deems dangerous.

While the server is characterizing the issue, the client looks for a single alternative configuration in which it can retry the problematic inputs. *Choosing an untried reconfiguration* can either mean requesting a reconfiguration from historically effective guards, or – if the client can afford to wait for the server – pausing until the new guard's suggestions are available. Once a reconfiguration is chosen, the client *reconfigures and retries* the failure-triggering inputs. Monitoring is still in effect, so if the same failure recurs, the client-side portion of the process may be repeated. (As long as the server optimizes away analysis of duplicate failures, the server-side portion will not repeat.)

The bottom row, with dashed arrows, depicts the proactive portion of Firefox-REFRACT. In the monitor phase, the client code *detects configuration or guard changes*, which may indicate that the client is now in violation of one of its configuration guards. Accordingly, it *consults those guards*, and, if necessary, *chooses a reversion* from those that the guards suggest. Then it immediately *reverts its configuration*, bringing its configuration back to a state the guards consider safe.

The flows in Figure 1 execute many times and for many Firefox-REFRACT clients. This will drive the system to accumulate a set of guards that characterize the failing configurations for a population of potentially diverse clients.

2.1 Building on Rainbow

REFRACT's use of the MAPE loop led us to to leverage an existing MAPE-oriented infrastructure – Rainbow [7]. Rainbow was designed to check the conformance of system execution against architectural and design assumptions. Rainbow is organized, as shown in Figure 2, into two layers: (a) a translation layer which is embedded into the target system and (b) an adaptation and strategy layer which is decoupled from the target. The translation layer distinguishes three components: *probes* monitor the target system to gather information, *gauges* measure that information to facilitate judgments at the adaptation layer, and *effectors* modify the target system's behavior. The higher-level layer captures relevant *knowledge* that is needed to drive adaptation. Typically Rainbow keeps a model of the execution environment and the system architecture; the latter defines how the target system can be restructured. The knowledge and measurements reported by gauges are *evaluated* to determine if adaptation is warranted, e.g., gauge measurements may lie below thresholds that trigger adaptation. If adaptation is required, a *manager* selects from a suite of adaptation strategies and tasks the *executor* with achieving an adaptation. We describe REFRACT's extensions to Rainbow in Section 3.

2.2 Finding Workarounds

An important component of REFRACT is finding workarounds. We discuss our prior work in this domain here. Figure 3 is a simplified feature model for the Firefox web browser. There are three optional features (*keyword.enabled*, *privacy.SanitizeOnShutdown*, and *browser.autofocus*). We also have a feature (*browser.startup.page*) that is required in all instances, but that must take only one of three values (1, 2 or 3).

Suppose the system has the following starting configuration. It has enabled both *privacy.SanitizeOnShutdown* and *browser.autofocus*, but has disabled *keyword.enabled.* In addition, *browser.startup.page* is set to 3. If a failure is encountered at runtime, we want to move to another configuration to avoid this problem. Garvin et al. [21,23] proposed a brute force *n*-hop algorithm, that tries each possible configuration *n*-hops away from the starting one. For instance, a 1-hop variant of this algorithm would first include *keyword.enabled*, leaving all other features as-is. Then it would remove this feature again and remove *privacy.SanitizeOnShutdown*, etc. At each step it would re-run the use case to determine if it now passes and keep track of all passing configurations. These are possible workarounds.

Assume that the setting of 3 for *browser.startup.page* is a faulty configuration option. Two passing configurations would be detected by the 1-hop algorithm. This would include two configurations with the original settings, but *browser.startup.page* set to either 1 or 2. Thus the setting of 3 is determined to be potentially faulty, and it can be guarded against it in future runs. Reconfiguring the browser to one of the two passing configurations allows execution to proceed from the failure. This single option configuration is a simple case, for illustration purposes, but configurations requiring as many as 5 option changes have been found in our studies [43].

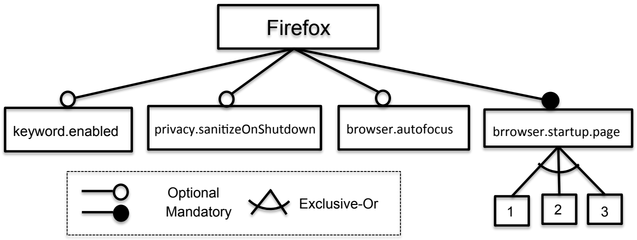

Figure 3: Example Feature Model

One of the advantages of a brute force approach is that we stay *close* to the original configuration and will maintain most system functionality from prior to the failure. In preliminary work [43], we found the 1-hop algorithm to be effective in avoiding a large number of failures. The 2-hop variant found additional workarounds, but only for failures that already had 1-hop solutions. However, scalability is limited. The *n*-hop algorithm grows exponentially with respect to *n*; 2-hop applied to a large configurable system was 233 times slower than 1-hop.

REFRACT is designed to run online and to be applied to clients with large complex feature models. Consequently, we explore the use of new configuration sampling algorithms to better balance cost-effectiveness in searching for configuration workarounds in the next section.

3. IMPLEMENTING REFRACT

In this section we first present REFRACT as pseudocode and elaborate on the sub-algorithms responsible for characterizing failures and suggesting workarounds. Then we detail how REFRACT is realized within Rainbow, building on the concepts in Figure 2.

3.1 Failure Avoidance in REFRACT

Algorithm 1 shows REFRACT at a high level. The system as a whole maintains a set of deployed clients (line 1), a feature model (line 2), and a set of guards (line 3); the client and guard sets are initially empty. When clients come online, their probes and gauges are initialized (line 4) and their guard sets are synchronized with the server's (line 5) before they are added (line 6). Likewise, probes and gauges are torn down (line 8) after clients are removed (line 7). Failures are handled by verifying replicability (lines 9 and 10) and calling a customizable sub-algorithm findGuards (line 11) (which we discuss in Section 3.2) to create new guards. The guard update triggers a reaction to the failure on the corresponding client, as well as proactive avoidance on other clients, via a test for guard-violating configurations (line 13). The sub-algorithm suggestions corresponding to the findGuards implementation produces possible ways out of such configurations, the best chosen according to a client-specific fitness function (line 14) and applied (line 15).

3.2 Sampling Techniques to Find Workarounds

For findGuards we have explored four different sampling strategies. One, outlined in [23], considers all configurations within a fixed number of hops (single-feature configuration changes) from the reported failing configuration. In this

Algorithm 1 REFRACT Failure Avoidance

```
Initially:
let $D \leftarrow \varnothing$ (the set of deployed client systems)
let $M \leftarrow$ the feature model
let $G \leftarrow \varnothing$ (the set of configuration guards)

When a client $d$ comes online:
instantiate an error probe and gauge on $d$
match $d$'s copy of $G$ to the server's
let $D \leftarrow D \cup \{d\}$

When a client $d$ goes offline:
$D \leftarrow D \setminus \{d\}$
tear down the error probe and gauge on $d$

When a client $d$ encounters a failure from inputs $I$ under configuration $C$:
replay $I$ under $C$ on the server
if a failure is observed then
  let $G$ (on both the server and the clients) $\leftarrow$
      $G \cup$findGuards$(I, C, M)$ (computed on the server)
end if

When a client $d$ changes to configuration $C$ or a client's copy of $G$ changes:
if $C$ violates a nonempty set of guards $G' \subset G$ then
  let $r \leftarrow$ a reconfiguration chosen from
      suggestions$(G)$
  apply $r$ to $C$
end if
```

strategy we systematically change all n-feature options one by one. Two more, *random sampling* and *covering array sampling* initially sample independently of the failing configuration (either randomly or using a covering array) and then, if a workaround is found, move to reconfigurations closer to the original via a reconfiguration minimization step. The last, a genetic algorithm whose fitness function aimed for configurations near the failing one, did no better than random sampling with minimization, and so is not presented here.

Random Sampling.

Random sampling repeatedly generates a random configuration from the feature model (ignoring the current configuration), resampling in cases where feature constraints are violated. In preliminary work [43], we experimented with this algorithm and found that small samples (around 10 configurations) found fewer workarounds overall than a one-hop sample, but occasionally found some workarounds that neither one- nor two-hop samples could find. While three times slower than the one-hop algorithm, random sampling was 118 times faster than two-hop exploration. Increasing the number of iterations up to 150, random found more workarounds overall than either one or two-hop sampling and still maintained an order-of-magnitude performance advantage over the two-hop approach. Thus, for the types of configuration spaces we experimented with, random seemed to strike a good cost-effectiveness trade-off.

Covering Array Sampling.

Covering arrays have been used extensively for testing highly configurable software systems [37, 45]. A covering array is a sample that ensures at least one t-combination of configuration options for all possible (valid) t-combinations in the configuration space. In the simplest case, t is 2, and such a sample is often called a pairwise sample.

For the covering array variant of our failure avoidance algorithm, we generate a t-way covering array for the feature model, and this constitutes our sample of workarounds to try. There has been research on combining delta debugging with covering arrays [28] and using covering arrays to find a minimal failing test [34], for which reason we believe that this approach makes sense. In our preliminary work [43], we found that 2-way covering arrays did not work as well as random sampling. But when we combined two different 2-way arrays together, more workarounds were found than with random sampling and in less time. Our best sampling strength for finding new workarounds consistently was a 3-way covering array, but this significantly increased cost. As with the number of iterations for random, here sampling strength exposed a cost-effectiveness trade-off.

Minimizing Workarounds.

Because the random and covering array sampling techniques can move a client arbitrarily far away from the configuration it started in, we have implemented a minimizer using the delta debugging technique [47], but adjusted to seek a minimal reconfiguration that passes, not fails, a test case.

To illustrate minimization, consider the feature model in Figure 3 and a failure caused by *browser.startup.page* set to 3. Suppose we randomly select a configuration where *privacy.sanitizeOnShutdown* and *browser.autofocus* are toggled and *browser.startup.page* is changed to 2. Since *browser.startup.page* must be 3 for failure, this configuration allows the system to pass the test case. Unfortunately, it is three hops from the starting configuration.

The minimizer divides the reconfigurations into two groups. In this example, the workaround the leaves a value of 3 for *browser.startup.page* from the starting configuration will fail; the other configuration will pass. In that latter configuration, some of the previously changed values were left untouched, so the hop count is decreased. The process is repeated until the minimal one-hop workaround is found. In this simple example, the minimization is actually more expensive than one-hop sampling. But as the feature model scales, sparse sampling and minimization scales better than the direct n-hop technique.

3.3 Modifications Made to Rainbow

REFRACT extends Rainbow to support detection of system errors and failures, their evaluation, and their avoidance via system reconfiguration.

REFRACT extends Rainbow's translation layer. New probes, P_{fail}, can either detect system failures, e.g., uncaught exceptions, oracle violations, or internal errors which might be caught as contract or assertion violations that remain enabled during deployment. This arrangement provides a means of avoiding failures by reconfiguring before errors propagate to output. New gauges, G_{fail}, assess the severity of a failure/error. For failures a binary measure might be appropriate, but finer measures may be appropriate for internal errors or when developers have information about the impact of classes of failure/error. Finally, configuration effectors, E_{config}, function by modifying configuration settings, e.g., in files, and performing micro-reboot of affected system components [1] or by modifying target configuration through programmatic APIs [26]. One of the major benefits of the effector is that the results of searching for a workaround, can be shared across all clients. This means that if one client encounters an error, and finds a workaround, another client will gain knowledge of the workaround and improved its guard. Sharing information allows for failure avoidance without needing a user to actually encounter the failure themselves.

REFRACT enriches the knowledge sources to include the target *feature model*, local preferences of *configurations to avoid*, and accumulated global information on *faulty configurations*. Unlike Rainbow, in REFRACT this knowledge is maintained by communication among multiple REFRACT instances; there can be a set of distributed communicating client-side REFRACT components.

To date, REFRACT has evaluated failures, $Eval_{fail}$, to determine the need for adaptation by applying an eager strategy – the presence of any failure gauge reading triggers adaptation. Failure evaluation that incorporates error/failure severity measures and history permits controlling trade-offs between the cost of reconfiguration and the exposure to future failures. This is also supported.

REFRACT might *select from a range of configuration strategies*. For example, REFRACT may also compute a safe configuration based on all target client failure reports that could be switched into for rapid failure adaptation. To date, REFRACT has focused on executing a *failure avoidance* strategy that seeks to find failure avoiding configurations that are close to the current target configuration. The intuition is that this will preserve behavior while avoiding failures. This behavior can also be modified.

4. CASE STUDY

In this section, we present a case study to evaluate the feasibility of REFRACT. We answer three primary research questions. Associated artifacts from this study can be found on our website.[1]

RQ1: How effective is REFRACT in finding workarounds for failures?

RQ2: To what extent can guards protect against previously seen failures?

RQ3: How does the number of system failures vary over time with REFRACT?

4.1 Client Program

We selected a highly configurable system for this study, Firefox version 18 [32]. Firefox is widely deployed, has a large bug database and automated testing framework and has been used in prior research on testing configurable systems [22]. It also has an automated testing framework, Mozmill [31], that comes with regression tests and can simulate clients running Firefox use cases.

We extracted the Firefox feature model (see our webpage for the complete model) by pruning from the configuration options found in the preferences page, `about:config`. Options removed from this set were the hardware-specific, security (we leave evaluation of security features as future work within a controlled sandbox), plugin, extension, and font options, as well as string- and integer-valued preferences for which a clear set of allowable values was not obvious, such as those that change storage space, or time delays. A core set of 311 options remained – 291 binary and 20 ternary. The total configuration space therefore has an order of around 1.4×10^{97}.

4.2 Client Use Cases

Each client use case in our study is based on a Firefox test case. We selected the *functional* category of tests from the most recent version of Firefox, version 26, and ran all 66 tests on Firefox 18. 30 did not pass due to requirements changes between the two versions; 36 applicable tests remained. To these we added seven that include assertions about which features are enabled, tests that give the same effect as a seeded configuration-dependent fault. Each assertion corresponds to a fault in version 18 whose report in the Firefox bug repository [33] included a reconfiguration workaround, some of which were taken from [22].

The full set of 43 test cases is on our website. As an example, one of the added seven is based on Firefox Bug 306208, in which Firefox displays a tab bar on popups when `browser.tabs.drawinTitlebar` is set to `true` (the tab bar should only be displayed on a main window).

4.3 Failure Avoidance Algorithms

In our initial study on GCC [43], the random algorithm and a 3-way covering array were the two top contenders when combined with the minimizer. We also found that the brute force one- and two-hop algorithms work well, but can be more costly time-wise. But our GCC feature model had only 166 options, and the test cases averaged less than 0.10 seconds, whereas Firefox has nearly twice as many options and tests taking roughly 20 seconds each. The hop-based algorithms would not be suitable when the clients wait for replies from the server. Therefore, we chose random sampling stopping at the first workaround found in up to ten iterations and a 2-way covering array with 23 configurations.

Additionally, our setup prefaces each server-side analysis with a loop over the already found workarounds. If one succeeds, then the effector will report that a workaround has already been discovered and bypass the failure avoidance algorithm.

4.4 Simulation Details

We implemented a prototype of REFRACT using four Firefox clients, each with its own instance of Mozmill, built on top of a version of Rainbow provided by the developers at CMU [41]. In the simulation the rainbow master starts first, followed by each of the clients. Once all of the delegates are instantiated, the gauges activate, and the monitoring begins. Whenever a Mozmill instance reports a failure, the corresponding monitor creates an error file and records the configuration under which that failure occurred. The file's update triggers the analysis phase, and the MAPE loop proceeds as described before.

[1] `http://cse.unl.edu/~myra/artifacts/Refract_2014/`

Algorithm 2 Algorithm for Running Simulation on Client

```
 1: let $C \leftarrow$ the current configuration
 2: let $G \leftarrow \varnothing$ (the set of guards)
 3: let $f \leftarrow$ false (a failure has not been discovered)
 4: while time remains do
 5:   update $G$ from the server
 6:   let $r \leftarrow$ a random single-option reconfiguration
          applying to $C$
 7:   if $G$ permits $r$ then
 8:     change $C$ by $r$
 9:   end if
10:   let $m \leftarrow$ a random Mozmill test
11:   let $f \leftarrow$ whether $m$ fails under $C$
12:   if $f$ (a failure has been discovered) then
13:     send $m$ and $C$ to the server and wait for a reply
14:     if a workaround configuration $c'$ was found then
15:       log $c'$ as workaround for $m$
16:     else if the failure could not be replicated then
17:       log the failure of $m$ under $C$ as not replicable
18:     else if a workaround could not be found then
19:       log that $m$ under $C$ has no current workaround
20:     end if
21:     let $f \leftarrow$ false
22:   end if
23: end while
```

We ran the simulation on a cluster of 1440 AMD CPUs running CentOS Linux. Each Firefox client and the master were run on different nodes and housed in separate directories to ensure independence of configurations.

Algorithm 2 shows the client-side portion of the simulation process. Each client keeps track of its current configuration (line 1), the known guards (line 2) and whether a failure has been encountered (line 3). To simulate gradual changes in client configurations, random single-option reconfigurations are periodically applied (line 6), though they may be prevented (and thus effectively reverted) by the known guards (lines 7–9), which are kept up to date (line 5). To simulate ongoing use cases, clients randomly select and run tests from the Mozmill test suite (lines 10 and 11). In case of a failure, the client opts to wait for a guard from the server (line 13), logging the corresponding workaround if it exists (lines 14 and 15), or the reason why one was not found if not (lines 16–20). The whole process repeats until the allotted time expires (line 4).

We note, that it is possible in a real scenario for a user to continue working once a failure has been encountered, however for the purpose of this simulation we wanted to keep the clients with failed tests isolated to ensure we can analyze the data with more accuracy. Future work will implement this additional aspect of the work. We are also only using a single master and one failure avoidance algorithm per run. This means that failures can only be addressed one at a time in the order they are received. This can cause performance bottlenecks, but the algorithm can be parallelized in the future.

The starting configuration is kept as default for one of the clients and randomly varied by up to five configuration options for the other three clients. We want to simulate what we consider a realistic user client base and do not think that real users will be very far from the default configuration.

Table 1: Count of workarounds found for the seeded failures by technique. Data is accumulated over five runs, except the 24 hour run which was run once.

Firefox Bug#	R10-3HR	R10-24HR	2CA-3HR
344189	5	1	5
306208	2	1	5
797945	5	1	5
808290	2	1	4
840411	3	1	3
442970	1	1	3
505548	0	0	0

As workarounds are found the configuration information is deployed to each of the clients, which allows them to update their own personal guard.

4.5 Metrics

To answer each of our questions, we mine the logs to keep track of the the run time, the errors encountered and whether a workaround is found. We also count the number of guards associated with particular failing combinations and map these back to the test that would have created that guard (guards are not necessarily related to a specific use case, but we can tell which use case created them). In addition, when a non-seeded failure is found, we evaluate whether or not it is a real failure/workaround (if not we consider this a false positive).

4.6 Threats to Validity

We have several threats to validity of this study. First, we have only run this on Firefox so we do not know if it will generalize. However, we have evidence from the prior work on GCC [23] that it behaves similarly with respect to a post-hoc analysis. Second, our faults are seeded and most are due to a single configuration option. But these are all based on real faults that have been reported in the field, and the last seeded fault (see our discussion of this threat later) is caused by the combination of two options. To reduce the threat that our implementation is in error, we re-ran experiments several times and three of the co-authors cross-checked results manually. Our data is available online so that the community can check our results as well. Finally, different metrics could have been used, but the ones we have selected address our research questions in a very direct way.

5. RESULTS

We ran simulations for both the random with ten iterations (R10), and 2-way covering array (2CA) workaround algorithms five times for three hours. We also ran a longer 24-hour run of the R10 algorithm. We use this data to answer each of our research questions in turn.

5.1 RQ1: How effective is REFRACT in finding workarounds for failures?

We first examine data from the seeded test cases (labeled by the Firefox bug number that they simulate). Table 1 shows the data from random failure avoidance with 10 iterations, on five 3-hour runs (R10-3HR) as well as a single 24-hour run (R10-24HR), and 2-way covering array failure avoidance on five 3-hour runs (2CA-3HR). For each run we evaluate whether or not it found at least one workaround and total across all five runs. For the 24-hour run we have

Table 2: Unexpected failures with workarounds for Mozmill tests. An FP indicates a false positive. The number indicates how many times this was found across all runs of the technique.

Mozilla Test#	# Workarounds (# FPs)		
	R10 3HR	**R10 24HR**	**2CA 3HR**
PlockupsBlocked	1	0	0
PopupsAllowed	2	1	0
ClearFormHistory	2	1	2
SSLDisabledErrorPage	1 (FP)	0	0
AddMozSearchProvider	1	1	0
OpenSearchAutodiscovery	2	0	2
RestoreHomepageToDefault.	0	1 (FP)	0
SubmitUnencryptedWarning	0	0	1
AutoCompleteOff	0	0	1
PasteLocationBar	0	0	1 (FP)
SetToCurrentPage	0	0	1 (FP)
EnablePrivilege	0	0	1

Table 3: Time in minutes to first workaround and to workaround not found for 3-hour runs. There are 23 configurations in the 2CA.

Sim Run#	Time to W	Time to N
R10-3HR 1	7.58	N/A
R10-3HR 2	7.95	N/A
R10-3HR 3	6.13	11.88
R10-3HR 4	8.43	7.32
R10-3HR 5	7.22	N/A
Avg. R10-3HR	**7.43**	**9.60**
2CA-3HR 1	8.82	14.78
2CA-3HR 2	14.78	49.38
2CA-3HR 3	10.62	15.98
2CA-3HR 4	9.93	N/A
2CA-3HR 5	12.92	19.24
Avg. 2CA-3HR	**10.20**	**24.05**

only one data point. For six of the seven bugs workarounds are found at least once. The 24-hour run using random reconfigurations finds six of these. The R10-3HR and 2CA-3HR always find workarounds for the first and third bug. The 2CA also finds workarounds every time for the second bug (306208). For the others, some of the runs find workarounds, and others do not. The covering array has slightly better consistency.

The last bug in the list (505548) never ran a test that failed during our simulations, and therefore it never looked for a workaround. In fact, in each instance where we did not find a workaround, it was because the test did not fail during the span of the simulation. Finding a failure requires the user to select a configuration and test combination that will fail. Like a long-running deployed execution, the longer simulation provided more chances for this to happen.

We examined bug 505548 more closely. First, this bug requires two configuration options to be used together before it will fail. Therefore, it may take longer to trigger a failure. We also saw that one of the configuration options that is required to make this test fail is in the guard created by bug 442970 – specifically the configuration setting of *browser.startup.page* as 3. This means that once bug 442970 has been seen and a workaround is found, bug 505548 will always be prevented from failing, so we should not expect to see it fail as long a 442970 has put this guard in place. This concurs with the results in Garvin et al. [23] on feature locality, i.e., that failures will be localized among a small number of feature options. Therefore, one failure guard should help prevent other failures as well.

In addition to the workarounds we found for the seeded failures, we also obtained a set of workarounds for tests that were not supposed to fail (these were from the passing Mozmill regression test suite). We examined each workaround (reproducing the tests several times) to determine if it is a true workaround (i.e. the test actually fails, and then the new configuration allows it to pass). In some cases, we determined that the original test should not have failed during a run, and its failure was an anomaly (for some reason it did not setup and run correctly). We mark these as false positives. In the case of a false positive, the first change to a new configuration will pass (since the test should not have failed anyway) and will create an unnecessary guard. We want to avoid these, since it might reduce the usable configuration space by guarding valid configurations. Table 2 shows the tests that failed and indicates whether or not they are real failures or false positives. Over all of our runs, we found workarounds for 12 tests that were unexpected; four were false positives. We also see an interesting locality result here as well. The guard for *AddMozSearchProvider* and *OpenSearchAutodiscovery* both fail with *bidi.direction* set to 2, and pass when the value is set to 1. Therefore, once we find one workaround, we guard against the other.

Table 3 shows the time to find a workaround (**W**) or the time taken to find no workaround (**N**) for the 3-hour runs in minutes. The top half of the table shows the time to workaround for each of the R10 runs with an average of 7.43 minutes. To determine that a workaround is not found we need on average 9.60 minutes. This can be a performance bottleneck and suggests the need for parallelism in the execution of tests for our samples; fortunately this is embarrassingly parallel so we would expect the R10 times to be reduced by a factor of 10 or better. If we now examine the data for the 2CAs we see that the average time to workaround is only slightly longer than the R10 samples. Considering that the CA has more than twice as many configurations, this is encouraging. Since we are stopping at the first workaround, we seem to be able to find a passing one within a similar amount of time. However, the time to workaround not found is 24 minutes on average. Again, this implementation bottleneck can be alleviated through parallelism where we would expect similar, or better, reductions than for R10 since there are more configurations here.

Summary for RQ1 Based on this data, we conclude that we can find workarounds for all but one of the seeded failures (but that one is guarded against) and in addition, we find 8 additional failures with workarounds that we can reproduce. The 2CA is slightly more effective than the R10 runs when the system runs for 3 hours. However, the time to find that a workaround is not possible with the current implementation is twice as expensive for 2CA than R10.

5.2 RQ2: To what extent can guards protect against previously seen failures?

Tables 4 through 8 show data for our simulations broken down into time blocks. We parsed the data for each of our 3-hour runs into 15 minute time blocks, and into 2 hour time periods for our 24-hour run. We show two runs for the R10-3-hour and 2CA-3hour runs and for the R10-24-hour run

Table 4: Random-10 3-hour, Run Two (Worst). 15 minute intervals shown. P - pass, F - failure, W - workaround, A - workaround already found, N - no workaround found, G - Guard.

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	# Guards
Expected to Fail													
344189	FW	2P	4P	5P	-	PG	4P	2P	8P	G7PG3P	3P	4PG2P	4
306208	P	2P	P	5P	5P	3P	3P	4P	2P	P	5P	5P	0
797945	-	FWP	3P2GP	2P	3P	5P	3P	5P	2P	6P	2PGP	PG5PG	5
808290	P	3P	P	4P	5P	P	P	3P	4P	4P	6P	P	0
840411	2P	4P	2P	3P	-	4P	6P	3P	8P	4P	2P	6P	0
442970	P	2P	4P	P	4P	P	3P	2P	5P	2P	2P	5P	0
505548	7P	7P	-	4P	5P	4P	4P	3P	2P	6P	6P	3P	0
Not Expected to Fail													
ClearFormHistory	4P	3P	4P	3P	FPW5P	3P	P	5P	P	3PG	4P	GP	2
PopupsBlocked	5P	3P	P	4P	P	PF2PWP	6P	3P	5P	6P	2P	5P	0
AddMozSearchProvider	3P	3PF	PW2P	P	4P	4P	7P	6P	5P	4P	4P	3P	0
Total Guards													11

Table 5: Random-10 3-hour, Run One (Best). 15 minute intervals shown. P - pass, F - failure, W - workaround, A - workaround already found, N - no workaround found, G - Guard

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	# Guards
Expected to Fail													
344189	FW	2P	2P	P	P	2P	3P	3P	G	3P	5P	3P	1
306208	F	FWAPG	3P	P	PG	4P	4P	4P	2P	PG	2P	6P	3
797945	FWPG	G	5P	3P	4P	2P	2P	2P	4P	3P	3P	PG	3
808290	–	P	P	2P	P	4P	P	PFW	2P	3P	P	–	0
840411	3P	2P	5P	FPW2P	3P	–	6P	P	PG	–	3P	PG	2
442970	3P	–	2P	P	2P	2P	2P	2P	P	3P	2P	4P	0
505548	–	3P	4P	3P	4P	2P	–	P	2P	2P	P	2P	0
Not Expected to Fail													
OpenSearchAutodiscovery	P	P	6P	5P	3P	2P	3P	2P	5P	PF	W3P	2P	0
SSLDisabledErrorPage	FP	PA	–	2P	3P	2P	2P	5P	4P	P	3P	3P	0
UntrustedConnectionErrorPage	FN	P	3P	4P	3P	P	2P	P	2P	3P	2P	2P	0
Total Guards													9

(other data is online and appears to be similar). We tried to pick one run that was best in terms of finding the most workarounds and one that was worst, for each technique. In these tables we use a P to mean that a test ran and passed, an F to mean that a failure occurred during that time, an N, to show that no workaround was found and a W to mean that a workaround was found. Finally, we use a G to represent that a guard was activated. We use numbers to summarize the times each occurred within that block of time. For instance, FW2G would indicate that during this period a failure was seen once, a workaround was found and the guard was activated 2 times. We only show data for the seven seeded faults and any additional unexpected failures with or without workarounds that were seen.

We have shaded the cell where the first workaround (or a workaround already found) is seen to help visually see the timeline. In most cases, once we find a workaround, we do not see a failure again. We often do see the guard activated at least once showing us that the guard is working. One anomaly in this data is seen in the first of the two 2CA tables – Table 6. For bug 344189 and bug 306208, workarounds are found within the first 30 minutes. In 344189 we also see a guard activated at 45 minutes. But we see a new failure with no workaround for these in the 2:15 and 2:30 time periods. We examined our logs and saw that these are both on the same client. We ran the failing tests and configurations manually several times, and we could not reproduce the failure. We do not know what caused this to happen, but we were able to confirm that these failures were not related to configurations that should have been guarded. Consequently, we consider these false failure reports.

We show the sum of guards activated for each test during the simulation in the last column. We see at least 3 and as many as 11 guard activations in a 3-hour period. In the 24-hour run we see 158 guard activations. This indicates 158 times that a client was prevented from entering a bad configuration.

Summary for RQ2 We can conclude that the guards are activated and appear to be preventing target applications from experiencing failures after a workaround was previously found.

5.3 RQ3: How does the number of system failures vary over time with REFRACT?

To answer our last research question, we examine Tables 4 through 8 again. We see that the number of failures is reduced over time (as the guards start to activate). In particular, for the 24-hour run, after 6 hours all of the workarounds are found for the seeded faults; one more workaround is found at 16 hours for an unexpected failing test that is difficult to expose. There are some failures (lower part of the table) for which we do not find workarounds, and these are the only remaining failures after 16 hours.

Figure 4 depicts the trends of these data aggregated across all 10 of the 3-hour runs. The trends are clear. Early in the runs failures occur, but they are reduced rather quickly and the activation of guards can be seen as the mechanism for achieving that reduction.

Table 6: 2CA 3-hour, Run One (Best). 15 minute intervals shown. P - pass, F - failure, W - workaround, A - workaround already found, N - no workaround found, G - Guard

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	# Guards
Expected to Fail													
344189	F	WP	2PG	5P	-	2P	3P	P	2PF3PNP	5P	5P	PG	2
306208	FW	P	P	4P	2P	4P	2P	3PFP	PN3P	3PG	3P	5P	1
797945	F	2F	W2A5P	2PGP	4P	2P	7P	2P	5P	2P	6P	2P	1
808290	P	P	3P	2P	FWP	P	2P	3P	4P	2P	5P	PGP	1
840411	FW	2P	-	G4P	GP	5P	3P	3PG	PG2P	G4P	GP	2P	5
442970	-	P	FWP	2P	5P	2P	P	P	3P	2P	6P	2P	0
505548	2P	-	2P	3P	2P	5P	4P	P	3P	3P	2P	4P	0
Not Expected to Fail													
ClearFormHistory	4P	P	2P	2P	3PF	3P	PN2PFP	PW3P	4P	PG	2P	4P	1
SubmitUnencryptedInfo	- -	P	6P	3P	P	6P	3P	3PF	PW	6P	3P	0	
Total Guards													11

Table 7: 2CA 3-hour, Run One (Worst). 15 minute intervals shown. P - pass, F - failure, W - workaround, A - workaround already found, N - no workaround found, G - Guard

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	# Guards
Expected to Fail													
344189	-	FW	-	-	P	3P	2P	2P	2P	3P	P	0	
306208	-	-	-	-	-	-	P	2F	WPA	GP	2PG	2P	1
797945	-	2FWA	-	-	-	-	3P	2P	3P	2P	P	3P	0
808290	2P	-	-	-	-	-	2P	P	-	2PF2P	WP	2PGP	1
840411	3P	-	-	-	-	-	P	2P	P	3P	2P	3P	0
442970	3P	P	-	-	-	P	2P	-	P	2P	-	2P	0
505548	-	-	P	-	-	-	P	P	P	3P	-	P	0
Not Expected to Fail													
AutoCompleteOff	P	-	2P	-	-	-	FPW3P	P	-	P	P	0	
DisableFormManager	3P	-	FN	-	-	-	2P	-	P	-	4P	2P	0
Total Guards													3

Summary for RQ3 We conclude that REFRACT's combination of failure-adaptive reconfiguration and guarding of configuration changes reduces the failures experienced by target applications over time. Moreover, this study suggests that the failure rate for configuration related faults drops relatively quickly over time and stabilizes with executions that are free of configuration related failures

6. RELATED WORK

There is a large body of research on both self-adaptive systems [3, 4, 8, 9, 12, 14, 16, 20, 24, 40], and on testing highly-configurable software [27, 29, 37, 38, 45]. Self-adaptive software came out of research on autonomic computing [18]. Early work focused on self-healing such as that of Dashofy et al. [12]. Other work aims to improve the adaptation process [17], particularly in dealing with uncertainty [14]. The work of Georgas et al. [24] looks at support fo architectural configurations, and Zhang et al. [48], and Garlan et al. [8, 19, 20] focus on validation. In the work by Elkhodary et al. [16] they adapt based on features. While these features are similar to ours, the adaptation uses quality of service attributes.

Rainbow is a framework that uses the MAPE loop and architectural constraints to adapt [7, 8, 19]. The work of [42], uses self-adaptation in software product lines, a type of configurable software system. The most closely related work on configuration-aware adaptation, is our own [21, 23], but we did not implement our techniques on a running system.

Fixing faults automatically is a related thread of work. Perkins et al. [35] deploys patches to running systems. There has also been work on adapting software to avoid failures using standard APIs [13] or using alternative but equivalent execution sequences [25]. Weimer et al. [44] finds faults in systems using test cases, and then uses an extended form of genetic programming to fix the code. Carzaniga et al [2–4], find workarounds which are code-level alternate library calls. This technique is similar to ours in its goal and the requirements to find a workaround. But our technique uses a feature model and is external (we do not touch the code). In recent work, Casanova et al. [5], use Rainbow to diagnose faults on web pages. This is closely related, however they are not using configurations as a workaround strategy.

There is also research on testing highly-configurable system [10, 11, 22, 27, 37, 39, 45, 45]. Many sampling techniques have been used for selecting a subset of configurations for testing such as covering arrays for sampling configurable systems [37, 45] and software product lines [36], Reisner et al. [38] use symbolic execution of configurable software systems to gather evidence that we have a much smaller feasible configuration space possible then previously thought.

Last, there has been work on distributed quality assurance [15, 30] that uses a central client to distribute tests and gather data on faults across many sites and locations. The work of Yilmaz et al. [45] uses this for fault characterization, but does not adapt to find new, passing configurations.

Our framework is unique in that we do not target the client code (but adapts externally), we use discrete failures to trigger adaptation, and we guard against future reconfigurations into known bad states by using the distributed nature of our system to improve community knowledge. Finally, we have built a working implementation that uses the MAPE loop to monitor, analyze, plan and execute.

Table 8: R10 24-hour, 2 hour intervals shown. Passes are not shown. F - failure, W - workaround, A - workaround already found, N - no workaround found, G - Guard

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	#Guards
Expected to Fail													
344189	2FWAG	–	G	2G	G	G	–	–	2G	–	G	2G	11
306208	2FWAG	3G	–	5G	3G	–	G	4G	2G	–	3G	2G	24
797945	2FWAG	2G	–	G	–	G	G	–	G	G	3G	-	11
808290	–	–	FWG	5G	2G	G	2G	–	3G	2G	–	2G	18
840411	–	FW	4G	G	–	–	–	–	–	–	G	-	6
442970	F	WG	G	4G	G	2G	2G	G	G	G	G	G	16
505548	-	-	-	-	-	-	-	-	-	-	-	-	0
Not Expected to Fail													
ClearFormHistory	FWG	3G	–	5G	G	G	G	3G	3G	2G	G	3G	24
PopupsAllowed	–	F	W2G	2G	G	4G	G	G	3G	G	2G	4G	21
AddMozSearchProvider	2G	FWF	A2G	4G	2G	G	–	3G	–	3G	3G	-	20
RestoreHomepageToDefault	–	–	–	–	–	–	–	FW	G	G	–	5G	7
Not Expected to Fail - No Workaround Found													
GoButton.js	–	–	–	–	–	–	–	–	–	–	–	FN	0
CloseDownloadManager	–	–	FN	–	–	–	FN	–	–	–	–	FN	0
AutoCompleteOff	–	–	FN	–	–	–	–	–	–	–	–	-	0
DisableFormManager	–	–	–	–	F	NFN	FN	–	FN	–	–	-	0
OpenSearchAutodiscovery	–	–	–	FN	–	–	–	–	–	–	–	–	0
DefaultSecurityPrefs	FN	–	–	–	–	–	–	–	–	–	–	–	0
SearchViaFocus	–	–	–	–	FN	–	–	–	–	–	–	-	0
SSLDisabledErrorPage	–	FN	–	–	–	–	–	–	–	–	–	–	0
HomeButton	–	–	–	–	–	–	FN	–	–	–	–	-	0
Total Guards													158

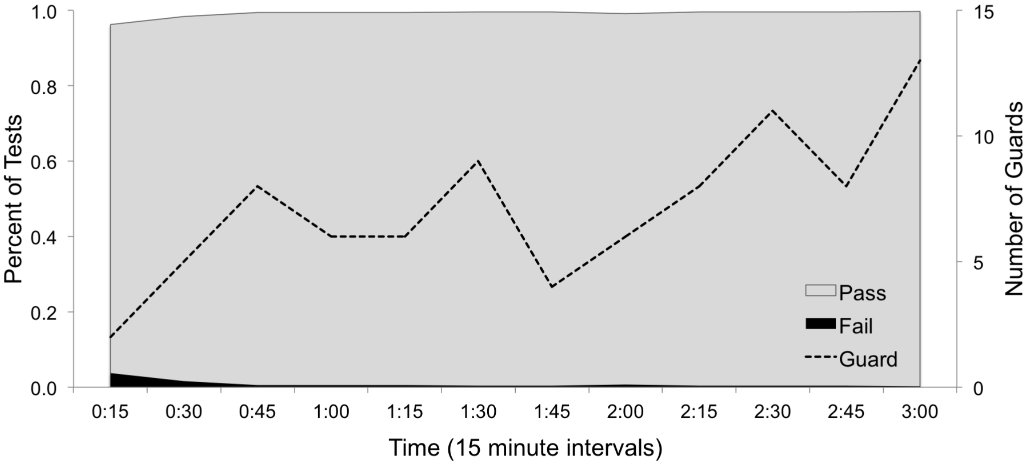

Figure 4: Percent passing and failing tests over time. Line shows number of guards activated. All data accumulated over 10 runs.

7. CONCLUSIONS

In this paper we have presented REFRACT, a self-adaptive framework for avoiding failures in configurable systems. REFRACT monitors the system for failures, and then triggers an algorithm to search for a passing reconfiguration, close to the original. It then updates a guard which prevents future clients from reconfiguring into known bad states. In a case study using 4 Firefox clients, we were able to find workarounds for all but one of our seven seeded faults and for eight native failures. The only missed seeded fault never manifested as a failure in our study. That missed fault was related to another, and by encountering the other failure, guards would be put in place to block clients from the configuration necessary to expose the missed fault.

In future work we will explore ways to parallelize the workaround algorithm so to use more clients and to come to a steady state sooner. We will also explore alternative algorithms, examine making the utility function continuous by weighting the error rather than using a binary value, and apply REFRACT to other client applications. Finally, we plan to incorporate user input into our decisions with respect to guards.

8. ACKNOWLEDGMENTS

We thank B. Schmerl for providing the Rainbow framework and support. This research was supported in part by the National Science Foundation awards CCF-1161767, CNS-1205472, and the Air Force Office of Scientific Research award FA9550-10-1-0406.

9. REFERENCES

[1] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *6th Symposium on Operating System Design and Implementation*, pages 31–44, 2004.

[2] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *International Conference on Software Engineering*, ICSE, pages 782–791, 2013.

[3] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE, pages 237–246, 2010.

[4] A. Carzaniga, A. Gorla, and M. Pezzè. Self-healing by means of automatic workarounds. In *International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS, pages 17–24, 2008.

[5] P. Casanova, D. Garlan, B. Schmerl, and R. Abreu. Diagnosing architectural run-time failures. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, pages 103–112, Piscataway, NJ, USA, 2013. IEEE Press.

[6] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. 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] S.-W. Cheng. *Rainbow: Cost-effective, Software Architecture-based Self-adaptation*. PhD thesis, Carnegie Mellon University, 2008.

[8] S.-W. Cheng, D. Garlan, and B. Schmerl. Making self-adaptation an engineering reality. In O. Babaoghu, M. Jelasity, A. Montroser, C. Fetzer, S. Leonardi, and A. Van Moorsel, editors, *Conference on Self-Star Properties in Complex Information Systems*, volume 3460 of *LNCS*. Springer-Verlag, 2005.

[9] S.-W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Shanghai, China, May 2006.

[10] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: A requirements engineering perspective. In *Theory and Practice of Software, International Conference on Fundamental Approaches to Software Engineering*, FASE, pages 16–30, 2008.

[11] M. B. Cohen, M. B. Dwyer, and J. 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.

[12] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS, pages 21–26, 2002.

[13] G. Denaro, M. Pezzè, and D. Tosi. Test-and-adapt: An approach for improving service interchangeability. *ACM Transactions on Software Engineering Methodology*, 22(4):28:1–28:43, Oct. 2013.

[14] A. Ebnenasir. Designing run-time fault-tolerance using dynamic updates. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2007.

[15] S. G. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[16] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *International Symposium on Foundations of Software Engineering*, FSE, pages 7–16, 2010.

[17] N. Esfahani, E. Kouroshfar, and S. Malek. Taming uncertainty in self-adaptive software. In *ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE, pages 234–244, 2011.

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

[19] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[20] D. Garlan, S.-W. Cheng, and B. Schmerl. *Increasing System Dependability through Architecture-based Self-repair*, pages 61–89. Springer-Verlag, 2003.

[21] B. Garvin, M. Cohen, and M. Dwyer. Failure avoidance in configurable systems through feature locality. In J. CÃąmara, R. Lemos, C. Ghezzi, and A. Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 266–296. 2013.

[22] B. J. Garvin and M. B. Cohen. Feature interaction faults revisited: An exploratory study. In *International Symposium on Software Reliability Engineering*, ISSRE, pages 90–99, 2011.

[23] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Workshop on Assurances for Self-adaptive Systems*, ASAS '11, pages 24–33, 2011.

[24] J. C. Georgas, A. van der Hoek, and R. N. Taylor. Architectural runtime configuration management in support of dependable self-adaptive software. In *Workshop on Architecting Dependable Systems*, WADS, pages 1–6, 2005.

[25] A. Gorla. Automatic workarounds as failure recoveries. In *Foundations of Software Engineering Doctoral Symposium*, FSEDS, pages 9–12, 2008.

[26] D. Jin, X. Qu, M. Cohen, and B. Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 31st International Conference on Software Engineering, Software in Practice (SEIP)*, ICSE, pages 215–225, 2014.

[27] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, jun 2004.

[28] J. Li, C. Nie, and Y. Lei. Improved delta debugging based on combinatorial testing. In *International Conference on Quality Software*, QSIC, pages 102–105, 2012.

[29] M. Lochau, S. Oster, U. Goltz, and A. Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Control*, 20(3-4):567–604, Sept. 2012.

[30] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. N. rajan. Skoll: Distributed continuous quality assurance. In *International Conference on Software Engineering, (ICSE)*, pages 459–468, 2004.

[31] Mozilla. Mozmill test repository.

[32] Mozilla. Firefox version 18.0, August 2012.

[33] Mozilla. Mozilla bug repository, 2013.

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

[35] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. 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.

[36] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Control*, 20(3-4):605–643, Sept. 2012.

[37] 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*, ISSTA, pages 75–86, 2008.

[38] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *International Conference on Software Engineering*, ICSE, pages 445–454, 2010.

[39] B. Robinson and L. White. Testing of user-configurable software systems using firewalls. In *International Symposium on Software Reliability Engineering*, pages 177–186, Nov. 2008.

[40] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous Adaptive Systems*, 4(2):14:1–14:42, May 2009.

[41] B. Schmerl. Rainbow implementation. `http://www.cs.cmu.edu/~able/research/rainbow/`, 2011. Carnegie Mellon University, personal communication.

[42] N. Siegmund, M. Pukall, M. Soffner, V. Köppen, and G. Saake. Using software product lines for runtime interoperability. In *Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, pages 4:1–4:7, New York, NY, USA, 2009. ACM.

[43] J. Swanson. A self-adaptive framwork for failure avoidance in configurable software. Master's thesis, University of Nebraska-Lincoln, 2014.

[44] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, ICSE, 2009.

[45] 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.

[46] P. Zave. Feature interactions and formal specifications in telecommunications. *Computer*, 26(8):20–29, Aug. 1993.

[47] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.

[48] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software Engineering*, ICSE, pages 371–380, New York, NY, USA, 2006. ACM.