Using Feature Locality: Can We Leverage History to Avoid Failures During Reconfiguration?

Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA
{bgarvin,myra,dwyer}@cse.unl.edu

ABSTRACT

Despite the best efforts of software engineers, faults still escape into deployed software. Developers need time to prepare and distribute fixes, and in the interim deployments must either tolerate or avoid failures. Self-adaptive systems, systems that adapt to meet changing requirements in a dynamic environment, have a daunting task if their reconfiguration involves adding or removing functional features, because configurable software is known to suffer from failures that appear only under certain feature combinations.

Although configuration-dependent failures may be difficult to provoke, and thus hard to detect in testing, we posit that they also constitute opportunities for reconfiguration to increase system reliability. We further conjecture that the failures that are sensitive to a system configuration depend on similar feature combinations, a phenomenon we call feature-locality, and that this locality can be combined with historical data to predict failure-prone configurations. In a case study on 128 failures reported against released versions of an open source configurable system, we find evidence to support our hypothesis. We show that only a small number of features affect the visibility of these failures, and that over time we can learn these features to avoid future failures.

Categories and Subject Descriptors

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

General Terms

Reliability, Experimentation

Keywords

Self-adaptive software, highly-configurable systems

1. INTRODUCTION

Self-adaptive systems are growing in interest as an alternative architectural model when the goal is to ensure continuous operation under a variety of environments [3, 5, 6, 11, 12, 19]. Adaptations force the system to reconfigure in order to increase reliability or performance and are triggered whenever environmental conditions necessitate change. The traditional adaptive feedback loop involves collecting and analyzing data, deciding what adaptation to use, and then acting on this decision by reconfiguring the system.

Recent research on *self-adaptation* and *failure avoidance* has developed techniques for preserving functionality in the face of deployed faults [12, 28, 33]. Work in these two areas has produced a variety of approaches: exploiting redundancy in the architecture or implementation [3, 5], directly modifying the source code [25,31], changing the architectural components or connectors [6, 11, 18], constructing adapters or wrappers to fix interoperability issues between components [12, 29], and transitioning to a set of precomputed "good" states when a failure is detected [13, 33].

While reliability, a concern of self-adaptive systems, measures the number of failures in a given time period, we might also focus on individual, discrete failures. A model for single failures is easier to build, because we need only replay failing test cases under various configurations. If different failures occur under similar configurations, then the combined model for a small number of failures will be a good approximation of the more general reliability model, and can be learned relatively quickly.

To explore the viability of this idea, we examine a closely related area of research, validating *highly-configurable systems*—systems with features that can be added and removed [7]. Such systems may contain faults that cannot be exposed under every choice of features, and the prohibitive cost of testing all choices means that these faults stand a higher chance of eluding testers [27, 32]. These types of systems, while not necessarily adaptive, have similar characteristics in that reconfigurations change the way executions occur while preserving a core set of functionality. For instance, FeatUre-oriented Self-adaptatION, (FUSION) [14], models adaptive systems in the same way we model highly-configurable systems, and it uses the notation of feature models [7] to represent adaptations. Moreover, in the FUSION case study, online learning is successful even though it identifies only a portion of features as relevant to the utility function. Correspondingly, there is some evidence in the software testing community that failures are dependent on only small combinations of features [19, 23, 24]

In this work, we build on the ideas of Hassan et al. [20] and Kim et al. [22], and hypothesize that failures have what we term *feature locality*, a tendency to depend on similar combinations of features. In consequence, configuration choices that avoid one failure are likely to avoid others and maintaining a history of failures will allow us both to avoid failures as we reconfigure as well as to select potential reconfiguration workarounds upon failure. We have evaluated our hypothesis on a case study crossing three releases of the GNU compiler collection (GCC), simulating online adaptations by reconstructing the failure time-line, and show that (1) very few features impact the visibility of any failure, (2) a knowledge of failure history increases our effectiveness more than eight times over uninformed reconfigurations within five reconfigurations, and (3) we see a downward trend over time in the number of reconfiguration attempts needed to achieve our goal. These results suggest that feature locality exists and may be useful for self-adaptive software.

The contributions of this work are:

- A presentation of feature locality and its potential impact on ensuring the dependability during reconfiguration.
- A case study to evaluate the existence of feature locality and our ability to exploit it on a set of failures detected in the field.

The rest of this paper is laid out as follows: In the next section we introduce background through a motivating example and discuss related work. We present our hypothesis in Section 3, Section 4 details our case study, and Section 5 concludes and highlights opportunities for future work.

2. BACKGROUND AND RELATED WORK

Our hypotheses and technique draw from two areas of software engineering: the research on dynamically reconfigurable systems, autonomic systems, and adaptation for correctness, and the work on prediction schemes for fault proneness. In this section, with the help of an example from NASA's Mars Exploration Program, we introduce the relevant terminology and illustrate the connections between this prior work and failure avoidance.

2.1 The Spirit Sol 18 Anomaly

Only 18 Mars solar days (sols, each approximately one Earth day) after landing, NASA's Mars rover, Spirit [1], encountered a nearly mission-ending software failure. The symptoms began with failed communications on the rover's two independent channels and eventually developed into intermittent, babbled transmissions along with an inability to obey basic commands. Not until sol 20 could NASA obtain crucial diagnostic information, including a health update packet, which showed signs of multiple reboots, a low battery, and a high internal temperature. Spirit was stuck in an endless reboot cycle and therefore failing to sleep; it risked running out of power or overheating.

The situation persisted through sol 21. Then the team at NASA managed a reboot with most of the flash file system disabled, and the rover started accepting commands consistently. Recovery became a possibility, though Spirit had to be put back in "crippled" mode every Martian morning.

In the meantime, NASA engineers strove to isolate the responsible software fault; although they were now able to avoid the failures, the root cause remained. On sol 71 the fault was found: a NASA-built component had expected the third-party file system to deallocate space as files were deleted, but deallocation only occurred when the enclosing directories were removed. As such, the file system had been bloated beyond a mountable size, and during boot, the failed mount from flash memory would trigger the default recovery action: a reboot. By sol 98 a fix was finished and installed.

In summary, the reconfiguration, which was found in three days through trial and error, meant that Spirit survived with limited functionality until the fix was delivered two and a half months later.

2.2 Terminology

Spirit was deployed as a single system, but its functionality was divided among *features*: the software and hardware components governing power, temperature, radios, sensors, motors, actuators, autonomous navigation, etc. Because most of these features could be enabled or disabled, Spirit constituted a *highly-configurable system.* However, like many highly-configurable systems, some features depended on or conflicted with others, so not every *configuration*—that is, not every choice of features—was valid. Under NASA's workaround for the anomaly, *feature constraints*, dependencies between features, disallowed most of Spirit's functionality because flash memory was disabled. (Even more functionality would have been lost without the workaround: Spirit would have been inoperable and eventually unrecoverable.)

A configurable system's features and feature constraints are usually represented compactly in a *feature model.* There are several languages for expressing a feature model in a form that mirrors the organization of functionality [2,10,26], but for our purposes we translate from these languages to a more uniform, relational form by following the process in [8]. The result is a sequence of configuration choices, each of which has a set of mutually exclusive options. For example, Spirit's solar panels and batteries, along with their associated software, can be toggled independently. Thus, the transformation creates one choice between active solar panels and a newly-introduced "null" feature, and it produces a similar choice for the batteries. We say that there are two *feature groups*, each containing two alternative features.

A sequence of independent feature choices makes for a structured model, but not one that is expressive enough to encode most feature constraints. Hence, as detailed in [9], we define one boolean variable for every feature and treat each configuration as an assignment to these variables; a variable is assigned true if and only if the configuration includes the corresponding feature. Then the restrictions on legal configurations can be expressed as a propositional formula. For instance, suppose we define P to mean that Spirit's solar panels are active and Q to mean that Spirit's batteries are providing energy. For Spirit to be in operation, $P \vee Q$ must hold, or else there will be no power supply.

Like other systems, when a highly-configurable system observably deviates from its requirements, we say that it has encountered a *failure* [21]. Spirit rebooting, nearly overheating, ignoring commands, and returning garbled data, for instance, constitutes a failure. We will reserve the term *fault* and its synonym *bug* for flaws in the system that make failures possible [21].

If the system would have met its requirements had it been configured differently, we say that the failure it encountered was *reconfiguration-avoidable.* In NASA's race to save Spirit, for instance, the rover began functioning correctly once engineers eliminated the flash memory mount.

Importantly, a failure may be reconfiguration-avoidable in some situations but not others; reconfiguration workarounds must sacrifice functionality to attain correctness. On sol 21, Spirit's goal was survival, so the loss of flash memory was acceptable. But if NASA had required the rover's more advanced scientific abilities to stay active, it is unlikely that the failure could have been avoided.

2.3 Adaptation for Correctness

For Spirit's development team, it was worthwhile to give special attention to a single deployment. That's not the case for most highly-configurable systems; we expect many deployments, perhaps in a variety of configurations. If we are to apply the lessons from the rover scenario, recovery and failure avoidance must be at least partially automated and, if possible, leverage information from other deployments.

Systems that employ such automation—those that have the capacity to monitor their environment and behavior and then react to more closely conform to requirements—are called *autonomic* or *self-adaptive* [28]. There is a large body of work describing such adaptation in both the hardware and the software domain. Researchers have considered systems designed to respond to poor performance [11], security vulnerabilities [25], architectural mismatches [11], misconfiguration [6,12], interoperability issues [12,29], and functional failures [4,6,13], all without human intervention. Additional studies have produced methods for validating these designs in safety-critical systems [30] and dynamic SPLs [19,24].

Throughout all of this work the adaptation process can be divided into four activities: *monitoring*, *detecting*, *deciding*, and *acting* [28]. Monitoring is responsible for sensing the program's environment and tracking its behaviors. The data it gathers feeds into detection code, which determines whether the observations should force a change in the system's state. If a modification is warranted, the decision routines are invoked to choose a response, in this case a reconfiguration. The action phase carries out the decision code's choice.

We are primarily interested in the decision phase of systems that combat functional failures, *self-healing systems* [3,5,6,11,12,19]. Once self-healing software has determined that a failure has occurred, it must decide how to restore the system to a known good state, retry the failed task, and avoid the failure on future tasks. We concentrate on the last two choices.

Our approach extends the intuition pioneered by Hassan et al. [20] and refined by Kim et al. [22]: just as memory accesses exhibit spatial and temporal locality patterns that can be exploited by a cache, fault-introducing changes to a system's source code also demonstrate locality. The latter work points out four forms of locality exhibited by faults, two that refer to the time of changes and two that refer to their locations in the source code. We hypothesize a related locality of failures (rather than faults) in a system's configuration space (rather than its change history or source code).

3. TECHNIQUE FOR AVOIDANCE

Our follow-on conjecture to this hypothesis is that we can learn from failures to guard against and recover from later ones. We view this process as an auxiliary function on top of the decision phase in the normal self-adaptive loop.

To evaluate the hypothesis, we make several simplifying assumptions. First, we assume that an explicit feature model is available to the software at runtime. We take as given the mechanisms to detect failures and to maintain state upon reconfiguration. Then, for the purposes of the study we target all reconfiguration efforts on avoiding the discrete failure events rather than rating configurations with a full utility function. However, we do capture some of the tradeoff between functionality and correctness that a real system's utility function would encode; we divide feature groups into those that can modified and those that must not change. Finally, we assume that the input stream can be broken up into segments, which correspond to the *test cases* that we use in the remaining sections.

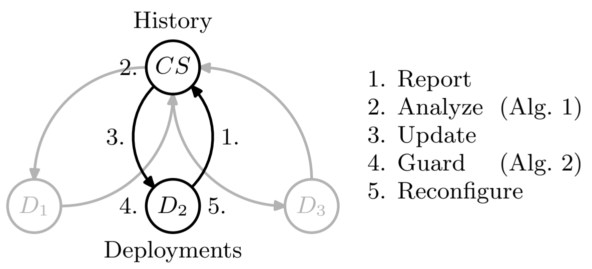

Figure 1: Exploiting History

Figure 1 illustrates our approach at a high level. Because it is black-box, our technique must learn about the system by encountering failures in the field. But it would be wasteful if every deployment of the system had to see each failure, so instead we establish a central store to relay failure information between deployments. In the figure, the store is labeled CS, and each deployment is designated by a subscripted D. Reporting of failures in deployments may be automatic or there may be some manual intervention. For instance, the reporting of a specific test case and configuration (below) may require a developer to provide additional information.

If we view a long running self-adaptive system as executing a sequence of phases where in each phase a specific configuration is used, then our method is triggered when a failure is detected, advises the reconfiguration process, and then allows the system to run under a new configuration. This process repeats when subsequent failures are detected.

Since we test our hypothesis on a system that is not self-adaptive, we simulate this type of behavior by having a single run of a program's configuration act like a phase in the execution of the long running system. For instance, when compiling a program one must choose a set of configuration options. If a compilation error, hard crash, or other failure is observed, one might then recompile the same program with a slightly different set of options until the failure is avoided.

Together, the central store and the deployment execute five steps. Step 1 begins the relay process: whenever a deployment detects a failure, it reports a test case and a configuration to the store. In step 2, an off-line analysis occurs to estimate whether the failure is configuration-dependent, and if so, which configurations it affects. For failures that have reconfiguration workarounds, the analysis results are broadcast in step 3. Step 4 uses this data to forbid configurations believed to be dangerous, both at the time the update is received and when a deployment is reconfigured for other reasons. Lastly, step 5 draws on the historical failure information to suggest workarounds when a new failure appears.

```
1  let $t \leftarrow$ the reported test case;
2  let $c \leftarrow$ the reported configuration;
3  let $d \leftarrow$ the maximum number of feature groups to
   change at a time (a parameter to the technique);
4  let $R \leftarrow$ the set of reconfigurations that affect between
   one and d feature groups;
5  foreach $r \in R$ do
6      let $c' \leftarrow c$ 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 configuration;
11         end
12     else
13         note $r$ as a possible workaround;
14     end
15 end
16 foreach $r \in R$ do
17     if $r$ is a possible or known workaround whose
       supersets in $R$ are all either possible or known
       workarounds then
18         note $r$ as a basis for generalization;
19     end
20 end
21 foreach $r \in R$ do
22     if $r$ is a strict superset of a basis for generalization
       then
23         forget that $r'$ is a possible or known
           workaround;
24         forget that $r'$ is a basis for generalization;
25     end
26 end
```

Algorithm 1: Analysis of Failures

```
1  let $c \leftarrow$ the current configuration;
2  let $T \leftarrow$ the set of test cases with known workarounds;
3  foreach $t \in T$ 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
7      foreach basis for generalization $r$ from $t$ do
8          let $c' \leftarrow c$ after applying reconfiguration $r$;
9          if $c = c'$ then
10             continue with the next iteration of the
               loop on line 3;
11         end
12     end
13     reject $c$;
14 end
15 accept $c$;
```

Algorithm 2: Guard on Configurations

Steps 1 and 3 are simply data transfers. The algorithmic steps 2, 4, and 5 warrant more detail, which we provide in the following subsections.

3.1 Failure Analysis

Step 2, the analysis of reported failures, is listed as Algorithm 1. The main idea is to try, by brute force, configurations that are similar to the one reported and see which ones pass. Although there are techniques to sample the configuration space more evenly [27], here we want the most accuracy near configurations that will see comparable inputs, presumably those that differ by only a few features.

Lines 1 and 2 begin by establishing the circumstances that led to the original failure, and lines 3 and 4 construct a set of reconfigurations to explore the vicinity. Throughout these algorithms we treat a reconfiguration as a set of choices for particular feature groups. For example, Spirit would have a reconfiguration to ensure that the solar panels are active, and a superset of that reconfiguration would guarantee that both the panels and the batteries are providing power.

Each reconfiguration is handled by an iteration of the loop on line 5. If, on line 7, the new configuration that results is both valid and suitable for the test case, the test is run by line 8 and the results recorded on lines 9 and 10. Otherwise, the algorithm notes that it could not evaluate the reconfiguration, at line 13.

Because the first loop only investigates within a small radius, the technique must make some generalizations to classify the rest of the configuration space. Our experience suggests that while failures may depend on several feature choices, the elimination of any one will usually constitute avoidance. Furthermore, if we are to preserve as much of the intended functionality as possible, we should favor small changes to the system configuration. Therefore, in the absence of contrary evidence, we generalize a workaround reconfiguration by assuming it to always mask the fault, even when other parts of the configuration are radically different.

The second loop, which begins at line 16, is responsible for determining which workarounds can be generalized without contradicting the algorithm's observations. Per line 17, an effective reconfiguration that is a subset of an ineffective one is not generalized.

Finally, the loop at line 21 discards information that is redundant in light of the generalizations. Specifically, if one reconfiguration is consistently effective, another reconfiguration that makes the same changes plus some extras is not useful; the conditional on lines 22–25 forgets it.

As an example, suppose that Algorithm 1 is applied to a failure that resembles the one Spirit encountered: the system misbehaves when all of four features, f_1–f_4 are present, and, moreover, f_4 is mandated by the test case. First, line 4 will calculate the set R, including a reconfiguration to disable just f_1, a reconfiguration to disable just f_2, etc. For each of f_1–f_3, the corresponding single-feature reconfiguration will cause the test case to pass, so all three will be known workarounds. The reconfiguration eliminating f_4 can't be attempted, so it will be labeled a possible workaround. Any supersets of these reconfigurations must either be impossible to test or also workarounds, so, regardless of d, these four are marked as bases for generalization on line 18. Their strict supersets are subsequently pruned by the loop at line 21, leaving the final diagnosis: any reconfiguration that disables one of f_1–f_3 should avoid the failure; reconfigurations that do not, but do disable f_4, might be effective.

3.2 Configuration Guard

Once the analysis results are available and distributed, Algorithm 2 guards deployments against dangerous configurations. For each test case where Algorithm 1 found feature selection to be significant, the guard checks that the current configuration is either known to be passing, on lines 4–6, or that a general workaround has been applied, on lines 7–12.

Continuing the example from Section 3, suppose that a deployment receives notification of the failure caused by f_1–f_4 while in a configuration that enables everything but f_2. Because Algorithm 1 identified the f_2-disabling reconfiguration as effective, and that reconfiguration has no effect on the deployment's current feature choices, it will assume that it does not need to take action. Similarly, if its configuration just disabled f_4, it could also continue, because there is a possibility that a workaround has been applied.

3.3 Choosing New Configurations

The last algorithm, for handling failures that the guard does not avoid, we only explain at a high level. First it builds a pool of candidate reconfigurations as in lines 3 and 4 of Algorithm 1. Then, after ruling out those that are not applicable to the current configuration and those that have been unsuccessfully tried before against the current failure, it chooses the one that suppressed the largest number of previous failures, breaking ties arbitrarily.

To complete the running example, consider the same deployment encountering a new failure, this time caused by the combination of f_1 and f_5. The pool of historically effective workarounds has reconfigurations to individually disable each of f_1 through f_3, and one of these—the one eliminating f_1—will succeed. In the best case it will be tried immediately, but in the worst case the failure won't be avoided until the third try.

3.4 Handling Multiple Versions

Complications arise when a new version of the system is released. The updated system may not have the same set of faults as the old one, and faults that do survive may have different reconfiguration workarounds, especially if the release contains new features.

In our experiments we track workaround data independently for each version. At release, the central store checks for all of the known failures and collects suitable reconfigurations for those that are found. Furthermore, when a failure is detected in one version, the other active versions are checked, if possible, under the same test case.

Under this policy forbidden feature combinations become available again as soon as the known faults are fixed. However, research also shows that new faults tend to appear in places where old ones were found [22]. It might be worthwhile to continue using data from old failures, especially if faults are fixed quickly, and so reduce the risk of failure at the expense functionality. We plan to investigate this tradeoff in future work.

4. CASE STUDY

As an initial evaluation to understand how configurations can be employed in avoiding failures we conducted a case study with 128 failures reported in the field for one of three versions of a highly-configurable software system. The study's research questions are presented in Section 4.1, and the systems, GCC is covered by Section 4.2. We describe our experimental methodology in Section 4.3, our threats to validity in Section 4.4, and we discuss the results in Section 4.5.

4.1 Research Questions

For our technique to be useful, there must be failures that it can work around. Hence, we first asked,

'existenceCan failures can be avoided by reasonable reconfigurations?

Provided that such failures exist, we must determine whether they exhibit feature locality:

'localityTo what extent do failures depend on similar combinations of features?

Table 1: Failures

	GCC	
Reported	**360**	**(100.0%)**
Incomplete	7	(1.9%)
Platform-Dependent	92	(25.6%)
Require Alternate Bootstrap Options	3	(0.8%)
Nonfunctional	13	(3.6%)
Nondeterministic	8	(2.2%)
Remaining	**237**	**(65.8%)**
Reproducible on Releases	**128**	**(35.6%)**

If the failures are present and localized, then we can ask about the effectiveness of our technique:

'avoidabilityCan feature locality be exploited to avoid failures?

4.2 Objects of Study

We evaluated our proposed technique on several versions of one highly-configurable software system, GCC[1]. Although there are characteristics of self-adaptive systems that it cannot capture, it is very representative of highly-configurable software and has some characteristics (described below) that we believe allows us to simulate such a system. Also, GCC has active user community and a public bug database, which we can mine for failures [16].

As in a distributed self-adaptive system, GCC deployments process separate streams of inputs (test cases), in this case sequences of compilation tasks including those in the bug reports, and operate under changing configurations (command-line options) for which we can extract a timeline from the bug database. Likewise, they are free to exchange failure information, here via human-generated reports for the GCC bug database (our central store). Because instances of the compiler are usually isolated from each other, we do not consider effects due to node interactions. Moreover, GCC reconfigurations can only happen between runs, so our evaluation does not capture behavioral changes during reconfiguration.

The following subsection describes this system in more detail, with an emphasis on how we obtained the feature model, failures, initial configurations, and so on.

4.2.1 GCC Versions

GCC [15], the cornerstone of the GNU toolchain, is a compilation framework with front-ends for a variety of languages and back-ends for a variety of platforms. The case study covers versions 4.4.0–4.4.2, all released in 2009, which each exceed 23 million lines of code.

In constructing GCC's feature model, we restricted ourselves to the compiler's command-line options, grouping features according the manual. We only included features that can be toggled without changing the input or the semantics of the output. One case deserves special explanation: GCC has some features that cannot be controlled completely from the command line. For example, the standard optimization

[1]We have used GCC (in addition to another application), and a nearly identical initial failure pool in a recent submission [17]. The only overlap with this work is that we employed Algorithm 1 (only described at a high level) to first determine whether a failure is configuration dependent. Beyond that, we performed a manual study of the faults at the code level for a different purpose.

packages (-O1, -O2, -Os, and -O3) enable some optimizations that have no corresponding flag. To handle these otherwise inaccessible behaviors we treated the use of each package as a feature, and put these pseudo-features in one group. If a failure depends on a hidden optimization from -O2, the technique will suggest workarounds like switching to -O1 and list the lost optimization flags explicitly.

We also assumed that several options are dictated by the test case being run: the stages of compilation to execute, the input language and its extensions (even when those extensions were not used), the platform or platforms being compiled for, the application binary interface, the set of enabled warnings, and the debug information that is emitted. This information is used by line 7 of Algorithm 1 and when selecting the new configuration to ensure workarounds preserve the user's intended behavior.

The complete model, including pseudo-features, totals 339 features in 168 groups, which means 171 single-feature reconfigurations are possible from any one starting point. All but one of the groups is binary (e.g. it has two possible options), while one has five possible choices. We also enumerated the dependencies among our features. In total we have 132 clauses to represent these constraints on the combinations of features in GCC. For example if we turn on the feature -fsched-spec-load, speculative motion of load instructions, then according to the GCC documentation we should also run instruction rescheduling before register allocation by enabling -fschedule-insns, O2, or O3.

For failures, we collected 360 reports from GCC's public bug database [16] that affect compilation or debugging for C, C++, and Fortran programs and are also tagged with "known to fail" on at least one of the versions in the 4.4.0–4.4.2 range. Then we chose an appropriate subset for the experiments:

First, we removed seven of these reports because they were still incomplete.

Then we discarded another 92 that depend on the platform where GCC is built, the platform where it runs, or the platform that it compiles for. Although we could have easily included failures that affect our platform—a 64-bit X86 system running openSuSE 11.0, which the auto-configuration detected as x86_64-unknown-linux-gnu—and we also could have used simulators to reproduce failures that call for other platforms, we were aiming to make our case study portably reproducible.

Next, because the bootstrap process that builds GCC is itself configurable, we further excluded three failures that required a non-default bootstrap configuration.

Finally, we omitted two other classes of failures: those where the problem is a violation of a nonfunctional requirement so we could not obtain an indisputable oracle (13 failures), and those that showed nondeterministic behavior (8 failures).

In summary, of the original 360 failures we kept nearly two thirds, 237. A synopsis of the failures excluded for various reasons is given in Table 1.

We then checked for each failure under every GCC version in our study. Almost half of the remaining failures were only visible in pre- or post-release revisions, so we could not reproduce them with the released code. On the other hand, of the failures we could reproduce, most affected all three versions despite being tagged with only one of the three as known-to-fail. The release of 4.4.1 showed only three failures that were not in 4.4.0; only two more were added from 4.4.1 to 4.4.2. The total line of Figure 2 shows the numbers for each version; the remainder of this table will be discussed in Section 4.5.

Finally, we used time stamps on the bug reports and the history of releases in the GCC SVN repository to build an overall picture of the sequence of events.

4.3 Methodology

For each failure, we are interested in the number of reconfigurations that avoid it and how many tries our technique needs to choose such a reconfiguration, in the best case and in the worst case. For each reconfiguration, we want to determine the number of failures it avoids. Because the association between failures and reconfigurations is already determined in Algorithm 1, we collect all of this data by simulating our technique.

The simulation must consider two types of events: the release of a new version and the discovery of a failure. It accounts for a release by running Algorithm 1 on every failing test case that is marked as seen before. To process a failure the simulation must follow a more complicated procedure.

First it loops through the versions that are deployed and applies Algorithm 1. Whenever the failure can only be achieved in configurations that Algorithm 2 would reject, the simulation notes that our technique would avoid the failure with zero reconfiguration attempts. Otherwise the failure is marked as seen.

For efficiency's sake the study only considers reconfiguration workarounds that change a single feature group; we set d in Algorithm 1 to one. As a consequence, the simulation may wrongly classify failures as unavoidable and bias the data against our technique. We also reran the experiment with d set to two, but the data remained identical. However, we cannot make conclusions beyond $d = 2$.

Next, for those versions where our technique cannot avoid the failure outright, the simulation determines the number of tries that the last algorithm would need to suggest a failure-avoiding reconfiguration. Ties in the sorting of reconfiguration attempts are broken to favor ineffective alternatives when we compute the worst case, and effective choices in the best case.

Finally, if the failure was seen in any version, the results of Algorithm 1 are saved. But if no version saw the failure, no data is kept for future use. There are two key assumptions here: First, we expect that the feature isolation process can be completed before the next failure is discovered. For GCC this requires only a few minutes—much less than the typical interval between bug reports. Second, we pessimistically permit multiple versions to encounter the same failure simultaneously.

4.3.1 Biased Random Reconfiguration

For comparison, we also simulated a technique that does not exploit the information gathered by the central store, but that uses other knowledge of the system which our technique does not have. It is meant to represent the approach to failure avoidance by reconfiguration that has an experienced user intervene when a failure occurs. First, the configuration space is not pruned, so every failure will be seen. Second, the attempted reconfigurations are chosen randomly, but with a bias towards workarounds that will succeed. The bias encodes the advantages of considering the type of failure, the input that triggered it, white-box knowledge, etc. We simplify the model by assuming that all ineffective reconfigurations have the same probability p, and that all effective choices have some probability q.

Because there is an element of randomness in this alternative technique, its worst case is to try the viable workarounds last and its best case to try them first. For the sake of a meaningful comparison, we consider its average case.

Let R be the set of candidate reconfigurations, $R^+ \subseteq R$ be the set of reconfigurations that will prevent a failure, and R^- be $R \setminus R^+$. If we assume—to the disadvantage of our approach—that the competing technique is not hampered by some configurations being illegal, the probability of it avoiding that failure within r reconfigurations is:

$$1 - \binom{|R^-|}{r} \Big/ \binom{|R^-| + \frac{q}{p}|R^+|}{r}. \qquad (1)$$

The calculation applies to exactly one failure, so the number of failures avoided in the average case is equal to this probability. Hence, for each test case we add the result of (1) to the avoidance count for the competing approach.

4.4 Threats to Validity

The major threat to its external validity is the fact that we only study one system which may not be similar to a real self-adaptive system. Although this system had a significant share of reconfiguration-avoidable failures, and these failures exhibited feature locality, further work is necessary to understand if these properties hold for self-adaptive software in general.

There is also the possibility that we only observed locality patterns because the failures in each system came from one source. It might be that the users who report bugs tend to use the compiler in similar ways.

Last, in our analysis of failures broken down by priority, we have a small number of faults in the highest priority category. This may limit our ability to make general conclusions.

For internal validity we must acknowledge the risks in the manual categorization and encoding of bug reports, the limitations of having only one test case to provoke each failure, and the ever-present risk of faults in our evaluation code.

Regarding construct validity there is a chance that the reconfiguration workarounds we propose may cause systems to encounter failures that are not in the bug database; in that case our reported rate of success will not be achieved.

4.5 Results

In the following subsections we discuss the results of our simulation in the context of each research question.

4.6 RQ1: Can Failures be Avoided by Reconfiguration?

Table 2 presents the reconfiguration workarounds we discovered, organized by failure and version. Each row presents the number of one-step reconfigurations that are possible. The first row shows the number of failures that had no workarounds at all, while failures with suitable configuration dependence are counted in subsequent rows. At the bottom we total the failures in each version and give the portion that have at least one known workaround, both as a count and as a percent of the total. For instance, in GCC 4.4.0 there are 95 failures with no workarounds and 31 that have at least one. If we examine the sixth row we see that there are six failures that have five one-step workarounds. This means that for each failure we can toggle any one of five features and the failure will no longer occur.

Table 2: Failures with One-Step Workarounds

# of One-Step	Counts		
Workarounds	GCC 4.4.0	GCC 4.4.1	GCC 4.4.2
0	95	87	80
1	9	9	9
2	4	3	2
3	3	3	3
4	3	3	2
5	6	5	5
6	2	2	1
7	2	2	2
8			
9		1	1
10		2	
11	1		
12			
13			
14	1		
Total	126	117	105
Nonzero	31	30	25
Percent Nonzero	25%	26%	24%

Table 3: Failures in GCC with Reconfiguration Workarounds, by Priority

	4.4.0	4.4.1	4.4.2
P1	3 of 5 (60%)	3 of 4 (75%)	2 of 3 (67%)
P2	7 of 23 (30%)	7 of 19 (36%)	5 of 17 (29%)
P3	21 of 84 (25%)	20 of 80 (25%)	18 of 75 (24%)
P4	0 of 11 (0%)	0 of 11 (0%)	0 of 8 (0%)
P5	0 of 3 (0%)	0 of 3 (0%)	0 of 2 (0%)
Total	31 of 126 (25%)	30 of 117 (26%)	25 of 105 (24%)

Roughly one quarter of the failures in each version are sensitive to reasonable reconfigurations, so the choice of features does play a significant role in a system's reliability. We also observe that, as in the study of Kuhn et al. [23], most of these failures are affected by only a few features—usually no more than six or seven. But we do see a handful of exceptions that can be avoided by changing any one of nine to 14 different features. In these cases there happens be a long data flow chain that invokes that failure and breaking it at any point prevents the failure's occurrence.

We next performed an analysis to see if high-priority failures have different characteristics than those with lower priorities. We used the rankings given by GCC developers to focus their fault-fixing efforts: from P1 (the most urgent) to P5 (unimportant).

Table 3 shows the breakdown. We see that none of the low-priority (P4 and P5) failures have reconfiguration workarounds, but a majority of P1 failures do which is encouraging. Percentage-wise we see a trend where failures with workarounds are more likely to be promoted from the default P3 status, meaning that the failures that matter most to developers are also avoidable by reconfiguration.

4.6.1 Summary of RQ1

In summary, approximately one quarter of the failures can be avoided by reconfigurations, with the fraction being larger for high-priority bugs (60–75% for GCC P1 reports).

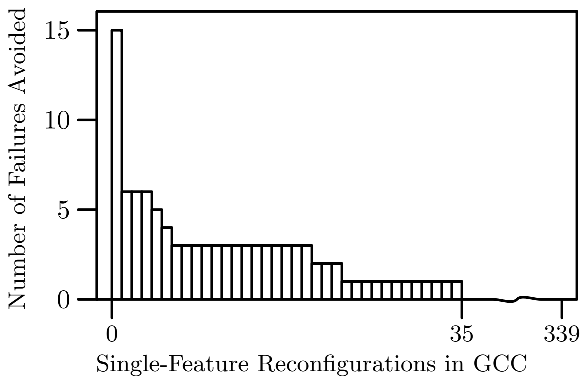

Figure 2: Feature Locality of Failures

4.7 RQ2: To what Extent do Failures Depend on Similar Combinations of Features?

Next, we view the same data from the perspective of reconfigurations to determine whether failures share workarounds. For each single-feature reconfiguration we counted the number of bug report/GCC version pairings where the reconfiguration avoided that failure. After sorting counts from highest to lowest, we plotted them in Figure 2. Note the broken scale on the x-axis, which elides 334 bars of height zero. This indicates that over 300 features had no impact on avoiding any failure. The spike on the far left corresponds to disabling undocumented optimizations by lowering the optimization level and then re-enabling the documented optimizations associated with the old level. This single reconfiguration avoids 15 failures on its own, but at the cost of the undocumented functionality.

We do see feature locality in this graph: less than 10% of the reconfigurations appear to affect correctness, and in an autonomic setting most of the feature choices would be free to vary in response to other concerns. Furthermore, the height of the bars shows that failures tend to have overlapping workarounds, and avoiding one failure often means avoiding others.

4.7.1 *Summary of RQ2*

Our data shows strong feature locality: a few reconfigurations have a significant impact on failure visibility, while for the remainder we detected no effect. Accordingly, in an autonomic setting, we need only consider a few features when predicting and reconfiguring for system reliability.

4.8 RQ3: Can Feature Locality be Exploited to Avoid Failures?

To answer RQ3 we compiled the simulation results to see if our technique was in fact effective. Figure 3 shows the data for the three versions of GCC on the left. The limit on the number of reconfiguration tries (25) is on the x-axis of each plot, and the number of faults avoided is on the y-axis (14-30). We shade the region between our technique's best and worst cases; its performance must fall in this region. For comparison, we also show the average case for random reconfiguration with various degrees of bias. The lowermost line is the expected behavior when effective and ineffective reconfigurations are equally probable, the next line makes workarounds twice as likely, and so on, until the topmost line, where failure-avoiding choices are preferred 64 to one.

For instance, on GCC 4.4.1, our technique avoids at least 19 failures within three reconfigurations, slightly better than we would expect from biased randomly chosen reconfigurations when the effective choices are 32 times more likely to be picked. In every case the technique prevents more than half of the reconfiguration-avoidable GCC failures from ever being seen, and the proportion increases in later versions because we retain information about surviving failures between versions. Using biased random reconfiguration as a ruler, our technique is nearly four times as likely to choose correctly after even 25 reconfigurations, and its performance matches much higher levels of bias earlier on. In short, GCC's feature locality makes historical workarounds good candidates for newly encountered failures.

We show another view of this data on the right hand side of Figure 3. The x-axis lists each of the 35 bug reports corresponding to reconfiguration-avoidable failures in chronological order. The intervals plotted against the y-axis give the best- and worst-case number of reconfiguration attempts needed to avoid each failure, with an interval omitted when the corresponding failure does not affect that GCC version. Note the break in the axis above 28 attempts. We had no prior information for some reconfiguration-avoidable failures and in these cases our technique could do no better than guess, which means at worst that it will try all 171 possible changes.

The main trend is captured in the plot for GCC 4.4.0: after an initial burst of learning from five failures that the technique is unable to avoid, the accumulated knowledge prevents nearly three out of every four failures outright, with several others avoidable in a handful of attempts. Meanwhile, in version 4.4.1, the patterns gleaned from 4.4.0 speed up the avoidance process, and in 4.4.2 everything is prevented except for two anomalies. Across all versions, 83% of failures are sidestepped without the technique resorting to guessing reconfigurations.

4.8.1 *Summary of RQ3*

Our simulation suggests that, because of the feature locality of failures, a system's failure history provides guidance for reconfiguration that effectively avoids new problems.

5. CONCLUSIONS

Self-adaptive systems have been studied widely for ensuring reliability. In this work we have examined self-adaptation for avoidance of individual failures. We leverage work from highly-configurable systems because these systems have faults that can only be exposed under some configurations. But, for the same reason, reconfiguration is enough to avoid these faults' failures in the field.

We conjectured that such failures exhibit feature locality, a tendency to depend on similar combinations of features, and we developed a multi-phase distributed algorithm to exploit this tendency. In a case study on a widely used system, we confirmed our hypothesis. We also showed that our technique can learn effective reconfiguration workarounds from early failures and use this information to guard against and avoid later ones. Even though it only had black-box information and was constrained by test cases that mandated some feature choices, our technique automatically circumvented one out of every four failures in the study.

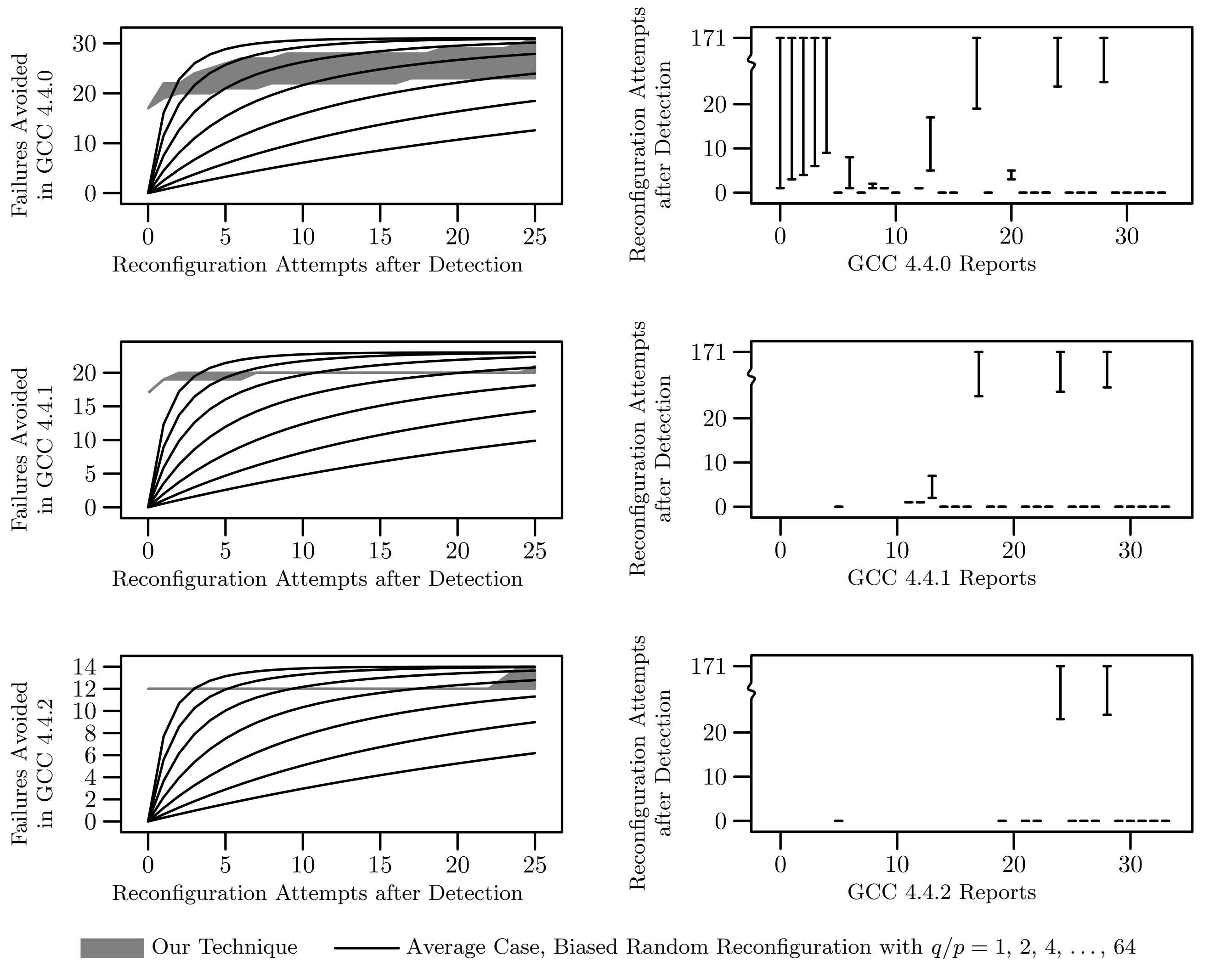

Figure 3: Number of Failures Avoided versus the Number of Reconfigurations after Detection

Furthermore, the strategy we proposed suggests several avenues for future work. We conservatively classified feature groups as either modifiable or mandated, but a more granular representation of the tradeoff between functionality and correctness could better capture users' preferences.

Finally, although GCC failures supported our conjecture, we have limited our study to a single system and simulated online adaptation. We plan to apply this idea to an online self-adaptive system next to understand how it will work in practice.

Acknowledgment

This work is supported in part by the National Science Foundation through awards CNS-0720654, CCF-0747009, CCF-0915526, and CFDA-47.076, by the Air Force Office of Scientific Research through awards FA9550-10-1-0406, FA9550-09-1-0129, and FA9550-09-1-0687, and the National Aeronautics and Space Administration through grant number NNX08AV20A. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of NSF, AFOSR, or NASA.

6. REFERENCES

[1] M. Adler. Mars exploration rover Spirit sol 18 anomaly. In *AIAA Space Conference/International Mars Conference*, Sept. 2004.

[2] D. Batory. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[3] Y. Brun and N. Medvidovic. Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In *Proceedings of the Workshop on Engineering Fault Tolerant Systems*, page 7, 2007.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[5] 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*, pages 17–24, 2008.

[6] H. Chang, L. Mariani, and M. Pezze. In-field healing of integration problems with COTS components. In *Proceedings of the International Conference on Software Engineering*, pages 166–176, 2009.

[7] P. Clements and L. Northrup. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[8] M. B. Cohen, M. B. Dwyer, and J.Shi. Coverage and adequacy in software product line testing. In *Proceedings of the Workshop on the Role of Architecture for Testing and Analysis*, pages 53–63, July 2006.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis*, pages 129–139, July 2007.

[10] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *International Software Product Line Conference*, pages 22–31, 2008.

[11] 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*, pages 21–26, 2002.

[12] G. Denaro, M. Pezzè, and D. Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–262, 2009.

[13] A. Ebnenasir. Designing run-time fault-tolerance using dynamic updates. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 15, 2007.

[14] A. Elkhodary, N. Esfahani, and S. Malek. FUSION: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, Nov. 2010.

[15] Free Software Foundation. GNU 4.1.1 manpages. Available at `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/`, 2005.

[16] Free Software Foundation. GCC Bugzilla. http://gcc.gnu.org/bugzilla/, Mar. 2010.

[17] B. Garvin and M. Cohen. Feature interaction faults revisited: An exploratory study. in submission 2011.

[18] 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*, pages 1–6, 2005.

[19] H. Gomaa and M. Hussein. Model-based software design and adaptation. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 7, 2007.

[20] A. Hassan and R. Holt. The top ten list: Dynamic fault prediction. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 263–272, 2005.

[21] IEEE Standards Board. *ANSI/IEEE Std 610.121990:Standard Glossary of Software Engineering Terminology.* IEEE, New York, NY, USA, 1990.

[22] S. Kim, T. Zimmermann, E. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

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

[24] F. Munoz and B. Baudry. Artificial table testing dynamically adaptive systems. Technical report, Institut National de Recherche en Informatique et en Automatique, 2009.

[25] 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 *Symposium on Operating Systems Principles*, pages 87–102, 2009.

[26] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering.* Springer, Berlin, 2005.

[27] 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.

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

[29] 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*, pages 1–7, 2009.

[30] E. Strunk and J. Knight. Assured reconfiguration of embedded real-time software. In *International Conference on Dependable Systems and Networks*, pages 367 – 376, 2004.

[31] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–374, 2009.

[32] 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.

[33] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software engineering*, pages 371–380, 2006.