# Feature Interaction Faults Revisited: An Exploratory Study

Brady J. Garvin and Myra B. Cohen
*University of Nebraska-Lincoln*
*Department of Computer Science and Engineering*
*Lincoln, NE 68588-0115*
*{bgarvin,myra}@cse.unl.edu*

***Abstract*—While a large body of research is dedicated to testing for feature interactions in configurable software, there has been little work that examines what constitutes such a fault at the code level. In consequence, we do not know how prevalent real interaction faults are in practice, what a typical interaction fault looks like in code, how to seed interaction faults, or whether current interaction testing techniques are effective at finding the faults they aim to detect.**

**We make a first step in this direction, by deriving a whitebox criterion for an interaction fault. Armed with this criterion, we perform an exploratory study on hundreds of faults from the field in two open source systems. We find that only three of the 28 which appear to be interaction faults are in fact due to features' interactions. We investigate the remaining 25 and find that, although they could have been detected without interaction testing, varying the system configuration amplifies the fault-finding power of a test suite, making these faults easier to expose. Thus, we characterize the benefits of interaction testing in regards to both interaction and non-interaction faults. We end with a discussion of several mutations that can be used to mimic interaction faults based on the faults we see in practice.**

***Keywords*-Interaction Testing, Configurable Software, Mutation Testing**

## I. Introduction

Configurable software systems, software systems with features that can be enabled or disabled, constitute an important class of software, one that is becoming more prominent. The software testing literature has paid special attention to configurable systems because, in addition to the kinds of faults present in other systems, they may contain faults caused by interactions of features [1]–[3]. These faults are termed *feature interaction faults*, or just *interaction faults* when the meaning is clear from context. Interaction faults can only be exposed under some combinations of system features [1]–[3] and the space of possible configurations is usually too large to exhaust (for instance, in GCC, the optimizer alone has roughly $10^{61}$ configurations [4]). Consequently, the research community has invested considerable effort in devising effective, but inexpensive testing methods [1]–[3], [5] with the most prominent being combinatorial interaction testing (CIT) [6], [7]. There is a growing body of empirical work on CIT with new methods and techniques to improve and extend interaction testing appearing rapidly.

Despite the extensive literature, the community still lacks a whitebox understanding of interaction faults. Researchers currently distinguish interaction faults from other kinds according to their blackbox behavior under a test suite [1]–[3], [5]. But test suites are rarely complete in practice, and such classifications are at best heuristic.

The lack of an exact distinction makes some fundamental questions about interaction faults difficult to answer. We do not know, for instance, how prevalent interaction faults are compared with other types, nor do we know if they are typically more severe or benign. While we can report the general fault-finding effectiveness of interaction testing techniques like CIT, we cannot know if the faults uncovered by these methods are the faults that are actually targeted. Neither can we seed interaction faults for experimentation; without such a definition it is unclear what mutations to use.

However, there is research that provides a good basis for a suitable whitebox view. In the work on feature interactions (originally developed for applications in the telecommunications field), researchers propose behavioral models for features and then apply model checking to detect system property violations [8]–[10]. And recent work by Reisner et al. [11] provides an analysis that links feature combinations to the structural coverage those combinations can add to a test suite.

We build on these results, working towards a theory of feature interaction faults for configurable software. First, we present a blackbox definition for interaction faults that is consistent with the literature. We then derive a necessary and sufficient whitebox criterion, where the condition is expressed in terms of code regions much like those identified in [11]. Then, in an exploratory study on hundreds faults from two open source systems we identify 28 that appear to be interaction faults at the blackbox level, and we check them against our criterion. Most do not match, and we shed some light on why they nonetheless seem to be sensitive to the system configuration as well as why techniques such as CIT might prove useful for finding them. We end with a discussion of the mutations that, applied according to our whitebox criterion, give a mutation strategy for seeding interaction faults.

The primary contributions of this paper are:

1) Necessary and sufficient whitebox conditions for the existence of an interaction fault.
2) An analysis of the distribution of interaction faults in two open source systems.
3) A better understanding of the benefit of interaction testing techniques for configurable software.
4) A mutation strategy for seeding interaction faults.

The remainder of this paper is laid out as follows: In the next section we present some background and related work. Section III follows with our blackbox definition and whitebox criterion, while Sections IV and V detail our exploratory study and its results. We conclude and present future work in Section VI.

## II. Background and Related Work

Our work extends the earlier research on testing configurable systems, drawing on the interaction testing literature especially [1]–[3], [11]. Because one of our goals is a strategy for seeding interaction faults, we also give background on mutation testing.

Throughout this paper, we will use the term *failure*, to represent any externally visible departure from a system's required functional behavior that is caused by the software. By *fault* (or sometimes *bug*), we mean a property of the system's implementation that makes such a failure possible. These are narrower meanings than are sometimes used in the dependability community (see [12] for more details).

### A. Configurable Software Systems

Configurable systems, systems in which features can be enabled or disabled, constitute a broad class, ranging from user-configurable products like web browsers to highly managed systems like software product lines [4], [13]. While they may be configurable at both the hardware and software level, we will restrict ourselves to configurable software. Configurable systems are organized into features, which Pohl et al. [13] defines as "end-user visible characteristics of a system." In this work, we view a *configuration* as a selection of features; it determines the presence or absence of every feature that could appear in a system instantiation. Not all configurations are valid; most systems mandate at least one common feature, and other limitations, such as the mutual exclusion of two features, are frequent [4]. Thus, we say that configurations are subject to *feature constraints*.

A *feature model* is a formal description of a system's features and the constraints that govern them [13]. Consequently, we can categorize a configuration as occurring at compile time (where the selection can usually be anticipated) or at runtime (possibly by end users, in which case configurations are harder to anticipate). Beside differences in who makes the configuration decisions, the time of configuration also changes how the system is analyzed. For example the work of Liebig et al. focuses specifically on software configured by preprocessor directives [14], whereas the techniques presented by Nita et al. concentrate only on runtime options [15].

A *configuration parameter* is an input—either to the build process or to the software system itself—whose purpose is to control whether one or more segments of code are reachable. It may be that some combinations of configuration inputs are illegal; for instance, they may lead to a failed compile, or they may be impossible to specify. It may also be that some combinations are equivalent, resulting in the same set of enabled features.

We also use the notion of a *staged configuration* in this work, where configuration choices are divided into *stages* [16]. Each stage's choices yield a more restrictive feature model, called a *specialization*, which is input to the next. For example, if we must decide on one of three alternatives, $A$, $B$, and $C$, the first configuration stage might eliminate $C$ but defer the choice of $A$ or $B$ to a later stage.

### B. Interaction Testing

In Figure 1 we show a snippet of code in which configuration parameters control which code is reachable based on a series of `#ifdef` statements. The important piece to note is that certain lines of code are reachable when single features are turned on or off while still other lines require more than one feature setting.

```
int main(){
 Data*data=readData(); // Line 2: Always reachable
 #ifdef FEATURE_A
  data=foo(data);      // Line 4: Reachable if FEATURE_A
 #else                     // is defined
  #ifdef FEATURE_B
   data=bar(data);     // Line 7: Reachable if FEATURE_A
  #endif                   // is not defined, but
 #endif                    // FEATURE_B is
 #ifdef FEATURE_B
  data=bar(data);      // Line 11: Reachable if FEATURE_B
 #endif                    // is defined
 baz(data);            // Line 13: Always reachable
 #ifdef FEATURE_A
  data=foo(data);      // Line 15: Reachable if FEATURE_A
 #endif                    // is defined
 return 0;             // Line 17: Always reachable
}
```

Figure 1. Identifying Feature Code

Research has shown that there are often specific combinations of parameter values that lead to faults in software [1]–[3]. For instance, in Figure 1 it is possible that the return value from the call to `foo` on Line 4, violates some assumption in the input space of `bar` on line 11 and causes an exception to occur. This data flow occurs only when both `FEATURE_A` and `FEATURE_B` are included in the program, but not when either one of them occurs on its own.

Some of the earliest work on interactions appears in the literature for telecommunications systems, where the possibility of such faults is called *the feature interaction problem*

[8], [9], [17]. Features such as `call waiting` may fail when combined with other features like `international calling`. In much of the research on feature interactions, models are built, often as state machines, and model checking is performed to verify if specific behaviors can be violated under the combination of specific features.

Out of this work grew much of the research on combinatorial interaction testing (CIT), which samples different combinations of parameter values for testing [6], [7]. The original work on interaction testing defined interactions as combinations of input parameters (rather than configuration parameters) where each full input is a test case. More recent work has expanded the study of interactions to the configuration level [1]–[3], [5], [10].

Interaction testing is typically a system-level, specification-based testing technique, so we usually examine variability from this view. Hence, we would model the presence of `FEATURE_A` and the presence of `FEATURE_B` as configuration parameters and describe their possible values as the set $\{\mathit{true}, \mathit{false}\}$.

When testing the software, we would run a set of system tests under different configuration parameter settings, such as $(\mathit{true}, \mathit{false})$ or $(\mathit{true}, \mathit{true})$. If we find that a fault appears only when both parameters take on the value *true*, we would call this an interaction fault—its presence depends on more than one option. Interaction faults are a special case of *configuration-dependent* faults, which appear under some, but not all configurations. For instance if there is a fault in the unit for `FEATURE_A`, it will only appear in the subset of configurations where `FEATURE_A` is set to *true*.

Empirical evidence shows that the blackbox behavior of interaction faults arises in many systems [1]–[3]. There has also been work to find the distribution on the number of configuration parameters required to detect interaction faults [2]. The count is generally small.

Yet there is a disconnect between the code and the interaction itself. The work of Reisner et al. [11] presents one definition of a feature interaction, but they restrict their work to interactions that can be expressed as structural coverage—that is, in terms of control flow. In the code of Figure 1 we must also minimally consider the data flow because it is the change of `data` that forces the fault in `FEATURE_B`. We set out to learn whether this is enough; we aim to understand what makes an interaction from a code-centric view.

### C. Mutation Testing

Mutation testing creates perturbations of a program that mimic faults a programmer might introduce. Testing is then used to detect (or "kill") these faults with the number detected used as a measure of the quality of the testing technique or test suite [18], [19]. Mutation can also be a method for automatically seeding faults for experimentation; Andrews et al. show that mutation faults are representative in certain respects [20].

There has been work on mutations specifically designed for integration testing [18], where the set of mutations focuses on unit interfaces in order to change the data that flows between sets of units. But because units may not map directly to features, or may be part of common code, this work is not specific to configurations or feature interactions.

## III. Towards a Theory of Interaction Faults

In this section we begin with a definition for an interaction fault and then present the necessary criterion for its existence. Intuitively, the essential point for an interaction fault is that it must involve more than one feature. If we are to give a formal definition, we must decide what involvement of a feature means. We will base our decision on dependence (in the probabilistic sense) and use staged configuration [16] as the framework for presenting our model. We also assume that we have a mapping between features and code.

Consider a two-stage configuration where we require every decision during the first stage to be of the form "feature $X$ will be present" or "feature $Y$ will be absent". This gives us a partial configuration:

**Definition III.1** (Partial Configuration)**.** For every feature, a *partial configuration* specifies one of three possibilities: that the feature is present, that it is absent, or that its presence is left undetermined.

Note that the constraints of the feature model may allow us to deduce the state of features not determined by the partial configuration. For instance, if feature $X$ requires feature $Z$, and the partial configuration indicates the presence of $X$, it does not matter whether $Z$ is declared present or undetermined. $Z$ could be also specified as absent, but then the staged configuration could not continue because all instantiations would be ruled out. Let us assume that such contradictions are disallowed.

We will define the size of this partial configuration as:

**Definition III.2** (Size of a Partial Configuration)**.** The *size* of a partial configuration is the number of features whose presence or absence it explicitly determines.

Before the first stage occurs we know nothing about the system instantiation apart from the requirements of the feature model. We can represent this knowledge by computing the system's *configuration space*, the set of all configurations that could be instantiated. Then, given the partial configuration, we can compute the maximal subset of the configuration space where the choices made in the first stage are honored. We say that these configurations *extend* the partial configuration. Because there are only two stages, the second stage must choose an element of this subset to instantiate.

Now suppose that the system contains a fault, visible under some configurations but not others. We call such a fault configuration-dependent, as a consequence of this definition:

**Definition III.3** (Configuration-Dependent Fault)**.** Given a test suite, a fault that it exposes under some configurations but not others is termed *configuration-dependent* with respect to that test suite [1]. If a test suite is not specified, it is understood to be the perfect test suite, which will reveal the fault under every configuration where doing so is possible.

In earlier work by Qu et al. [1] configuration dependence of individual test cases within a test suite is differentiated from configuration dependence for the full test suite. We do not make this distinction.

Next, consider the case where the fault is detectable in any extension of the partial configuration. We cannot conclude that the second-stage choices never affect the fault's visibility—it might be that some of them matter when the features are divided among stages differently—but we can conclude that the decisions in the first stage are enough to make them irrelevant. In that case, we would label the partial configuration as sufficient for exposing the fault:

**Definition III.4** (Sufficiency of a Partial Configuration)**.** A partial configuration is *sufficient* for exposing a fault if and only if for every full configuration that extends it there is a fault-exposing test case.

There will typically be many partial configurations that are sufficient; for every sufficient partial configuration that does not determine the presence or absence of a feature $X$, a partial configuration identical except for its ruling on $X$ will also be sufficient. However, we are only interested in minimum sufficient partial configurations, because for any fault contained within a single feature—which should not be an interaction fault according to the intuition presented earlier—there is a size-one partial configuration sufficient for exposing it. A natural step then is to use the size of the minimum sufficient partial configurations to distinguish interaction and non-interaction faults. Accordingly, we introduce the following two definitions:

**Definition III.5** ($t$-way Fault)**.** A fault is a *$t$-way fault* if there is a partial configuration of size $t$ that is sufficient for exposing the fault, but none of size $t-1$.

**Definition III.6** (Interaction Fault)**.** A fault is a *interaction fault* if it is a $t$-way fault where $t$ is at least two.

At first glance this last definition looks much like a characterization of integration faults, faults that require the integration of two or more units before they can be exposed. It differs in two main ways. First, units partition code, whereas features do not necessarily: code that connects features $X$ and $Y$ cannot be said to belong to either, and if

```
void foo(){
  if(FEATURE_A){            // r1; {c1,c2,c3,c4}
    bar();                  // r2; {c3,c4}
  }
}
void baz(){
  if(FEATURE_B){            // r1; {c1,c2,c3,c4}
    quux();                 // r3; {c2,c4}
  }
}
void main(){
  foo();                    // r1; {c1,c2,c3,c4}
  baz();                    // r1; {c1,c2,c3,c4}
}
```

$$\begin{array}{lcll} c_1 & = & \{\texttt{FEATURE\_A} \leftarrow \texttt{false}, & \texttt{FEATURE\_B} \leftarrow \texttt{false}\} \\ c_2 & = & \{\texttt{FEATURE\_A} \leftarrow \texttt{false}, & \texttt{FEATURE\_B} \leftarrow \texttt{true}\} \\ c_3 & = & \{\texttt{FEATURE\_A} \leftarrow \texttt{true}, & \texttt{FEATURE\_B} \leftarrow \texttt{false}\} \\ c_4 & = & \{\texttt{FEATURE\_A} \leftarrow \texttt{true}, & \texttt{FEATURE\_B} \leftarrow \texttt{true}\} \end{array}$$

Figure 2. Simple Example Configurable System where Instructions are Labeled with the Region they belong to and the Configurations that Make them Reachable

that same code could also connect $X$ and $Z$, then it is not even possible for us to give a unique set of owners. Second, units can be tested individually, while features might be implemented as conditionals scattered across many functions and unable to stand on their own.

Nonetheless, there is some mapping between features and the implementation, which we can use to guide our search for interaction faults or as an aid in seeding them. We begin by recalling the definition of a basic block:

**Definition III.7** (Basic Block)**.** A *basic block* is a maximal sequence of statements that has one entry point, one exit point, and no internal branching.

Grouping basic blocks according to their configuration dependence results in a set of variability regions:

**Definition III.8** (Variability Region)**.** A *variability region* is a maximal set of basic blocks such that if one of the basic block is executable under a given configuration, the other blocks are can be executed under the same configuration, though possibly by different non-configuration inputs.

Note that basic blocks that are interrupted by compile-time configuration guards, like `#ifdefs`, must be subdivided so that compile-time and runtime variability are on equal footing.

Take Figure 2 as an example. Like the previous example, the system has two features, `FEATURE_A` and `FEATURE_B`, and no restrictions on how those features can be combined. Thus, there are four configurations, enumerated as $c_1, \ldots, c_4$ at the bottom of the figure. Ignoring the callees, which we elide, each line lies in exactly one basic block and is labeled with the set of configurations under which some input can reach it. Lines labeled with the same set constitute one of the variability regions $r_1$ through $r_3$.

While the feature flags themselves are referenced in the example code, a configuration might influence reachability less directly, in which case labeling blocks becomes a more difficult task. But in systems where configurations are evaluated at runtime, we can approximate the labels by marking configuration parameters as symbolic and then executing a test suite, following the strategy presented by Reisner et al. [11].

Then, given a partial configuration, we can determine the corresponding specialization of the code, in which some variability regions may be guaranteed reachable or unreachable. For instance, given `FEATURE_A` $\leftarrow$ `true`, the regions $r_1$ and $r_2$ merge and are assured reachability. On the other hand, $r_3$ is still optional. With a different assignment, `FEATURE_B` $\leftarrow$ `false`, $r_1$ is reachable, the reachability of $r_2$ is unknown, and $r_3$ becomes dead code.

If the system in Figure 2 contains an interaction fault, an assignment to `FEATURE_A` alone cannot force the fault to be detectable, nor can a lone assignment to `FEATURE_B`. Therefore, in both cases the visibility of the fault must depend on code whose presence could still be in question—code outside of $r_1 \cup r_2$ on one hand and external to $r_1 \cup r_3$ on the other.

We must take some care, however, because we cannot conclude that the fault's visibility is always dependent on this code. For the moment we will note the regions that were made reachable and unreachable and add a caveat to our conclusions. Thus, we associate the pair $(r_1 \cup r_2, \varnothing)$ with `FEATURE_A` $\leftarrow$ `true` and the pair $(r_1, r_3)$ with `FEATURE_B` $\leftarrow$ `false`. $(r_1, r_3)$, for instance, means that code outside of $r_1 \cup r_3$ can alter the visibility of the fault when $r_1$ is known to be reachable and $r_3$ is known to be unreachable. A similar argument for the remaining size-one partial configurations, `FEATURE_A` $\leftarrow$ `false` and `FEATURE_B` $\leftarrow$ `true`, produces $(r_1, r_2)$ and $(r_1 \cup r_3, \varnothing)$, respectively. We call these pairs non-interaction pairs:

**Definition III.9** (Non-interaction Pair)**.** A *non-interaction pair* for a fault is a pair of sets of basic blocks $(R_1, R_2)$ such that, if the reachability of $R_1$ and the unreachability of $R_2$ guarantee the existence of a fault-exposing input, the fault is at most 1-way, and therefore not an interaction fault.

For each size-one partial configuration we can compute a non-interaction pair by finding the set of configurations extending the partial configuration. In the first coordinate we take the union of all variability regions whose labels are supersets of this set; in the second we have the union of all variability regions whose labels' complements are supersets—just as in the example. We say such a non-interaction pair is *induced* by the partial configuration.

As another example, suppose that we are analyzing a fault in Figure 3 triggered by `foo2()` but masked by `quux2()`. The assignment `FEATURE_A` $\leftarrow$ `true` gives us the set $\{c_3, c_4\}$ as possibilities. That set is a subset of the labels for

```
void main2(){
  if(FEATURE_A){          // r4; {c1,c2,c3,c4}
    foo2();               // r5; {c3,c4}
  }
  if(FEATURE_B){          // r4; {c1,c2,c3,c4}
    bar2();               // r6; {c2,c4}
    if(FEATURE_A){        // r6; {c2,c4}
      baz2();             // r7; {c4}
    }else{
      quux2();            // r8; {c2}
    }
  }
  xyzzy2();               // r4; {c1,c2,c3,c4}
}
```

$$\begin{aligned}
c_1 &= \{\texttt{FEATURE\_A} \leftarrow \texttt{false}, & \texttt{FEATURE\_B} \leftarrow \texttt{false}\}\\
c_2 &= \{\texttt{FEATURE\_A} \leftarrow \texttt{false}, & \texttt{FEATURE\_B} \leftarrow \texttt{true}\}\\
c_3 &= \{\texttt{FEATURE\_A} \leftarrow \texttt{true}, & \texttt{FEATURE\_B} \leftarrow \texttt{false}\}\\
c_4 &= \{\texttt{FEATURE\_A} \leftarrow \texttt{true}, & \texttt{FEATURE\_B} \leftarrow \texttt{true}\}
\end{aligned}$$

Figure 3. More Complicated Example Configurable System where Instructions are Labeled with the Region they belong to and the Configurations that Make them Reachable

$r_4$ and $r_5$, and a subset of the complement of the label for $r_8$. Thus `FEATURE_A` $\leftarrow$ `true` induces the non-interaction pair $(r_4 \cup r_5, r_8)$. `foo2()` is in $r_5$, and `quux2` is in $r_8$, so we can guarantee the reachability of the former and simultaneously the unreachability of the latter. Thus, the fault is not an interaction fault.

Clearly, non-interaction pairs are useful only if we can determine how the reachability of basic blocks affects fault detectability. Therefore, we define an analog to non-interaction pairs:

**Definition III.10** (Critical Pair)**.** A pair of sets of basic blocks $(S_1, S_2)$ is *critical* to a fault if and only knowing the elements of $S_1$ to be reachable and the elements of $S_2$ to be unreachable guarantees the existence of a fault-exposing input, apart from any other knowledge of the system configuration.

In other words, a critical pair $(S_1, S_2)$ and a non-interaction region $(R_1, R_2)$ where $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$ is enough to show that a fault is not an interaction fault. In fact, we can extend this claim to necessary and sufficient condition for an interaction fault:

**Theorem III.11** (Necessary and Sufficient Condition for an Interaction Fault)**.** A fault is an interaction fault if and only if there is no critical pair $(S_1, S_2)$ and size-one partial configuration $P$ such that $P$ induces the non-interaction pair $(R_1, R_2)$ with $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$.

*Proof:* To show necessity, suppose that we have a $t$-way interaction fault (with $t \geq 2$) and also $S_1$, $S_2$, and $P$ meeting the criterion above. Because $t - 1$ is at least one and necessarily less than the number of features, we can choose a partial configuration $P'$ that extends $P$ and also determines the presence or absence of exactly $t-1$ features. Any configuration that extends $P'$ also extends $P$, and we know that under such a configuration the basic blocks in $R_1$ are reachable while those in $R_2$ are not; the reachability of $S_1$ and the unreachability of $S_2$ follow immediately. But then we are guaranteed the existence of a fault-exposing test case, so that $P'$ is a size $t-1$ partial configuration sufficient for exposing the fault, a contradiction. Therefore, no interaction fault can violate this condition; it is necessary.

For showing sufficiency, we will demonstrate that the condition cannot hold for a non-interaction fault. By Definition III.6, we know that for every $t \geq 2$ there is a partial configuration of size $t-1$ sufficient for exposing the fault, which means that there must be a sufficient partial configuration $P$ of size zero or one. Let $(S_1, S_2) = (R_1, R_2)$ be the non-interaction pair induced by $P$. We have $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$ by construction, and the sufficiency of $P$ implies that $(S_1, S_2)$ is a critical pair. Consequently, the condition cannot apply to non-interaction faults, meaning that it is a sufficient criterion for interaction faults. ■

We recognize that the identification of critical pairs is not a trivial task. However, to demonstrate a non-interaction fault it is enough to find a one suitable critical pair, and it need not be minimal. With knowledge of the code change that constituted the fault, we can often identify all statements that might divert execution from indicted code, as well as statements that could alter the data it depends on. Then, between the fault and the point of failure we can collect the statements that might derail a failure. The faulty code and the point of failure then become $S_1$, while the other statements we identified become $S_2$. As long as we err towards over-approximation, the result will be a critical pair. And if the over-approximation doesn't include too many variability regions, the critical pair may be small enough to complete the argument.

On the other hand, to show that a fault is an interaction fault we can find statements whose execution is necessary for the failure, and possibly statements whose non-execution is necessary—statements that are guaranteed to mask the fault. Apart from code in the commonality, any critical pair must mention such statements in its reachable and unreachable sets, respectively. Consequently, we may be able to show that every critical pair's sets cannot be enclosed by the sets of a non-interaction pair, even though we cannot enumerate the critical pairs.

In our exploratory study we used both of these strategies to apply our definition to faults from the field.

## IV. Exploratory Study

Our study's research question aims to find out what portion of configuration-dependent faults are interaction faults according to our whitebox criterion in Theorem III.11:

**RQ: How well do configuration-dependent faults in the field match our criterion for an interaction fault?**

To answer this question we identify faults that are configuration-dependent and have been fixed. Based on the fix, we bound the minimum size of a sufficient partial configuration. In addition, we analyze the code to determine if it matches the sufficiency criterion and learn about the types of mutations that would represent interaction faults. This study does not use an automated implementation of our criterion, but rather, a detailed manual analysis. We leave the development of a fully automated analysis as future work.

### *A. Objects of Analysis*

We selected two open source highly configurable software systems for study: GCC and Firefox. Both are widely used, have publicly available bug databases with good documentation of the test cases and configurations that provoke each failure, and make commits and developer comments publicly available. They have different architectures, which allows us to draw slightly broader conclusions. For instance, features and units are aligned in GCC, while in Firefox features are implemented as conditionals scattered across many units.

GCC[1] is a compilation framework with front-ends for a variety of languages and back-ends for a variety of platforms. The study covers version 4.4.0, which exceeds 23 million lines of code. Importantly, GCC features line up closely with compiler passes and therefore with functions.

In constructing GCC's feature model, we restricted ourselves to the compiler's command-line options, grouping features according the GCC manual. This led to 168 features, most of which are binary (see our associated website[2]).

Firefox[3] is a leading web browser managed by the Mozilla Corporation. It is written in C, C++, and JavaScript, along with several other domain-specific languages, and the latest version contains more than 17 million lines of code. We considered every version of Firefox in the Mozilla bug tracker, but, because of that choice, many test cases had to be run manually. To save on human effort, we toggled only the features that the bug reports mentioned as significant when we checked for configuration dependence. Therefore, we did not have a 'global' configuration model, though we did restrict ourselves to the options on Firefox's about:config page. The full list can be found on our website.

### *B. Method*

To answer our research question we developed a set of qualitative metrics and then summarized them quantitatively. We categorized the underlying fault by asking four questions: (1) Could the fault have been caught by applying unit testing to individual features?; (2) Could the fault have been detected by precondition checks on functions/methods?; (3) Is the fault due to too much or too little information flowing between different features?; and (4) Would the wrong behavior have been correct for a different system configuration? Once we understood the fault well enough to answer these

[1]http://gcc.gnu.org/bugzilla/

[2]http://www.cse.unl.edu/~myra/artifacts/issre2011/

[3]https://bugzilla.mozilla.org/

Table I
SUMMARY OF STUDIED FAULTS

| **GCC version 4.4.0** | | | |
|---|---|---|---|
| **Total** | **Run on our system** | **Config. Dep.** | **Fixed** |
| 360 | 137 | 31 | 17 |

| **Firefox (all versions)** | | | |
|---|---|---|---|
| **Total** | **Fixed** | **Config. Dep.** | **Run on our system** |
| 118 | 116 | 11 | 11 |

questions, we put the fault into one of four categories, which were synthesized from the data, and applied our whitebox criterion. We then compiled mutation descriptions for the true interaction faults.

*Selecting Faults for Study.* For GCC, we collected all 360 reports from the public bug database 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 (many of the reports not tagged with 4.4.0 were still reproducible under that version). Then we chose an appropriate subset for the experiments, excluding reports that (1) were still incomplete, (2) required a non-default bootstrap, (3) described a fault that was fixed before the public release, or (4) could not be reproduced on our system. 137 remained. We ran each on our system, varying every configuration input individually; if some single change to a configuration parameter masked the fault, we marked the fault as configuration-dependent. 31 were so marked. There were only fixes on file for 17 of them, so we discarded the rest. Table I summarizes this data.

Extracting bugs from the Firefox database proved more difficult. The Firefox developers use it to track not only bugs, but also other tasks like routine maintenance (bug #598795, for instance), and publicity events (such as bug #262292). After some trial and error, we decided to focus on bugs marked as regressions and priority 1 (the highest priority), as these were almost always reports of functional failures. We included bugs from all versions of Firefox, in order to have a sufficient pool—118 reports. We then removed two unfixed bugs and read each report looking for mention of configuration dependence. 11 remained, all of which we manually reproduced on our system, varying the mentioned configuration parameters one by one to determine if we could mask the fault. All 11 were confirmed to be configuration-dependent.

*Detailed Analysis* Once we obtained a set of configuration-dependent faults for each subject, we manually examined the commits that fixed each of the faults, along with the developer comments on the issue trackers. When necessary, we also observed the faults in a debugger. After each inspection, we wrote a detailed description of the fault; these descriptions are cataloged on our website, along with links to the `diffs` for the relevant commits.

As a check of validity, we provided another researcher who was neither involved in this project nor aware of our

Table II
CONFIG. DEPENDENT FAULTS: DETAILED DATA

| **Sys.** | **Bug #** | **Parameters** | **Category** | **Interaction Fault?** |
|---|---|---|---|---|
| GCC | 39794 | 14 | II | Y |
| | 40087 | 3 | I | Y |
| | 40321 | 5 | IV | N |
| | 40389 | 2 | IV | N |
| | 41016 | 1 | IV | N |
| | 41094 | 1 | IV | N |
| | 41183 | 4 | IV | N |
| | 41403 | 3 | IV | N |
| | 41643 | 4 | I | Y |
| | 41843 | 1 | IV | N |
| | 41917 | 2 | IV | N |
| | 42049 | 6 | IV | N |
| | 42231 | 6 | IV | N |
| | 42542 | 4 | IV | N |
| | 42614 | 3 | III | N |
| | 42667 | 2 | IV | N |
| | 43024 | 4 | IV | N |
| Firefox | 306208 | 1 | IV | N |
| | 337871 | 1 | IV | N |
| | 344189 | 1 | IV | N |
| | 403040 | 1 | III | N |
| | 413437 | 1 | III | N |
| | 414836 | 1 | III | N |
| | 442970 | 1 | III | N |
| | 479994 | 1 | IV | N |
| | 529667 | 1 | IV | N |
| | 403854 | 1 | IV | N |
| | 423960 | 1 | III | N |

objective a random ordering of the 28 reports and asked him to independently do the same analysis on as many faults as he had time for. In total we checked 25 of the 28 faults (including all of those deemed interaction faults).

## C. Threats to Validity

The first threat to this work is that we only studied two open source subjects. We did select systems from popular application classes and with very different characteristics and architectures, but we may have missed important phenomena. Another threat is the possibility of human error because the study involved a manual analysis in unfamiliar code. In addition, one of the humans doing the analysis was an author. Although our conclusions were verified by another researcher, there may still be mistakes or unintended bias. We have therefore provided a detailed description of each fault analysis on our website, along with links to the issue tracker artifacts, so that other researchers can verify our results. Finally, the mechanism for checking configuration-dependence may be a threat: in GCC we only toggled one feature at a time, and in Firefox we relied on reporter or developer comments. Therefore, we may have underestimated the population of configuration-dependent faults in both subjects.

## V. RESULTS

In this section we answer our research question and discuss observations obtained from the faults in this study.

## A. *Configuration Dependent Failures*

Table II presents the results for the 28 faults that we determined to be configuration-dependent. The first column gives the software system, and the next shows the report's number in the corresponding issue tracker. The column labeled *Parameters* gives the number of configuration parameters that we could toggle to mask the fault—for GCC and Firefox configuration parameters are generally in one-to-one correspondence with features. For instance, bug #41183 in GCC 4.4.0 requires four specific feature settings in order for the fault to appear under the given test case.

A "1" in the parameters column is some evidence that a fault is not an interaction fault; however, we did not test the entire configuration space, so it is not proof. One half (14 of 28) fit into this category. For the bugs that seem to need at least two features enabled, we see the values range from two up to 14 with an average of 4.4 (or 3.7 if we remove the outlier of 14). And the median count is four, so assuming the availability of test cases like those reported, we can expect testing techniques that target interactions of four or fewer features to capture a large number of these problems. On the other hand, we did see four faults (roughly 14%) that need five or more features under these test cases.

The last two columns present the category of fault, which we describe next, and whether or not we would label this as a true interaction fault based on the definitions from Section III. After manually analyzing each of the faults we derived four categories that apply to all: two for the interaction faults and two for the non-interaction faults. They are as follows:

I. **Violation of one feature's assumption by another.** This is what we expected to see for most of the interaction faults: an exchange of information, about which features made inconsistent assumptions. A tester could target such faults with an analysis to identify information flow between features.

II. **Features fail to exchange enough information.** In contrast to the previous category, we also found a fault where intended information exchange was missing, something that would be difficult to identify by a whitebox analysis.

III. **The wrong features are enabled or they are enabled at the wrong time.** Although we knew that such faults could exist, this was not a category we expected to see so frequently. Here, the code implementing the features is correct, but the translation from configuration parameters to a configuration was incorrect. We also include faults that caused unintended runtime reconfigurations, and one case (GCC bug #42614) where the correct optional code is enabled but then called at the wrong time.

IV. **Fault in optional feature.** In the final category we collected all of the faults that were contained in functions implementing a single optional feature.

Table III
CATEGORIZATION OF FAULTS

| No. | Category | Subject | Count |
|---|---|---|---|
| I | Violating feature assumption | GCC | 2 |
| | | Firefox | 0 |
| II | Features fail to exchange info | GCC | 1 |
| | | Firefox | 0 |
| III | Wrong features enabled | GCC | 1 |
| | | Firefox | 5 |
| IV | Fault in optional feature | GCC | 13 |
| | | Firefox | 6 |

Table III summarizes the categorization by system. We see that out of the 17 GCC faults examined, only three are true interaction faults (category I or II): two of type I and one of type II. The rest were either faults contained and triggered entirely in optional features (category IV) or, in just one case, due to incorrect configuration-realizing code (category III).

Interestingly, while the majority of GCC configuration-dependent faults were actually non-interaction faults (13), many required a large set of features to be exposed by the given test case—bugs #42049 and #42231 for instance each need six features. We will discuss this phenomenon more in the next section.

None of the faults in Firefox fit our definition of an interaction fault. While all appear to be configuration-dependent, almost half were due to the application entering an incorrect configuration (5 of 11). And like GCC, the majority (6 of 11) were faults in an optional feature. While GCC and Firefox have different distributions of configuration-dependent faults, in both cases most fell into category IV, and less than a quarter belonged to other categories.

We summarize our answer to our research question as follows. We found that only about 10% of the faults appearing to be configuration-dependent are actually interaction faults, based on our earlier definition and criterion. Most were actually faults contained within a single optional feature (where $t = 1$) and in theory they could have been caught by more intensive feature testing. After that, many were due to incorrectly instantiating the intended configuration. But we do see some real interaction faults for GCC and believe that this is some evidence in support of interaction testing.

## B. *Mutation Categories*

We next discuss some mutation categories for the three faults that fall into the first two categories—the interaction faults. More detail is on our website.

*GCC Bug #41643.* In the simplest GCC interaction fault, an `if` that should have tested a condition in the form `(!foo||!bar)` was wrongly implemented so that it only checked `!foo`. Although the change is a simple first-order mutation, it occurred in optional code, and the predicate could only be falsified by statements in code for different features. The mutated statement, a use, and one of its defs were in the reachable blocks of every critical pair.

The general mutation suggested is to identify a viable def/use pair that is not contained by any non-interaction pair's reachable set, and then to apply a normal mutation to either the def or the use.

*GCC Bug #39794.* As with bug #41643, the mutation corresponding to GCC Bug #39794 is semantically a guard condition changed to `true` or `false`. The fix is actually somewhat more complicated because the GCC developers had to move one definition (which represents the canonicalized expression for a memory address) earlier and then change a function signature in order to reference the correct values in the guard.

The effects of this mutation are similar to the effects of #41643, but the reasons for it being an interaction fault are different. Although the `if`'s use is in code for an optional feature (a dead store elimination pass), it can draw on a wide variety of defs, not all of which are in optional code. The fault is only an interaction fault because the subsequent assignment almost always leads to equivalent gen and kill sets and therefore the same ultimate outcome. We identified at least twelve features that would disturb the inputs to the dead store elimination pass enough to mask the fault.

*GCC Bug #40087.* The most complicated of the three bugs was #40087, which was arguably several faults caused by a single misunderstanding. In five places, and in four different ways, the mutation altered a guard, affecting the value escaping a function in optional code; at the same time, all of the code that could use the def was governed by a different set of features.

The unique aspect of #40087 is in how the guards changed. At two points the correct condition to test is `false`. It might be better to classify those mutations separately, as introducing a definition. For this kind of mutation to yield an interaction fault, the location of the injected definition and the locations where it could be read must not be within the reachable set of any non-interaction pair.

*Summary.* We observe that most of the mutations are simple guard mutations that impact a def/use pair between variability regions. Therefore, an analysis to identify non-interaction pairs, an analysis that finds def/use pairs, an analysis that identifies associated guards, and an inventory of branch condition mutations may be enough to mimic these faults. We propose such a tool for future work.

## C. Discussion and Observations

Only three of the faults in Table II proved to be interaction faults according to our definition. The remainder could be detected at the function level, at the feature level, or with 1-way CIT, provided that we had a suitable set of test cases. Though surprising, this distribution still argues for the use of CIT or other configuration-aware testing techniques, as noted in the observations that follow.

*Interaction Testing Improves Feature-Level Testing.* Although many faults were contained in a single feature, and could have been detected by better feature-level testing, the test cases that would find these faults were often not obvious.

For example, GCC bug #41843 fails because common code mistakenly declares two structures equivalent when one's fields are a subset of the other's. At the system level, we would have to have arranged for these structures to be compared against each other by the faulty function and in the right order—an unlikely event if we did not have reason to suspect this particular bug. But when GCC performs structure-peeling optimizations (which create versions of a structure with fields removed), a fault-triggering situation becomes more plausible.

This points to an interesting observation: although interaction testing may not be necessary for finding such faults, the variation of enabled features generates a broader set of inputs for the feature, exercising behaviors that may otherwise go untested.

*Configuration-Realizing Code Needs to be Tested.* A large number of the faults found in Firefox fell into Category III, which we describe as the wrong configurations enabled. This was unexpected, but it may be because Firefox's features do not line up well with units, so the configuration-realizing branches must appear in many places, and are therefore harder to test. The prevalence of the category points out the need to test the configuration manipulation code more carefully. Although the faults of this type were not interaction faults, they may be hard to detect without a technique like CIT that samples the configuration space.

*Some Interaction Faults are due to Missing Information Flow.* At the start of this study we expected that we would be able to develop a static analysis to pinpoint potential interaction faults. However, as we examined the real data we came across category II, where the problem is nonexistent data flow or missing control flow between features. Without an oracle that describes the intended information exchange, this type of interaction fault is only detectable by combining black- and whitebox techniques.

*Overall Observations.* We now summarize our overall observations:

- Better feature-level testing and tests of the configuration-realizing code are important.
- Some interaction faults do exist in the field, so techniques that find them and mutations that represent them are still necessary. Because we observe a case where a whitebox analysis based on identifying data and control flow would miss the interaction fault, a system-specification-based technique such as CIT may be a good fit.
- In light of the low incidence of interaction faults and the high prevalence of blackbox configuration dependence, it seems that the real benefit of CIT is its ability to magnify the fault-finding power of a test suite.

## VI. Conclusions

In this work we revisited the notion of a feature interaction fault, presented a blackbox definition, and gave a necessary and sufficient whitebox criterion for an interaction fault to exist. We performed an exploratory study on two open source systems to understand if real-world faults fall into our more stringent definition of an interaction fault and to understand the types of mutations that would mimic these faults. Of the more than 250 faults that we considered, 28 were deemed configuration-dependent and formed the basis of our in-depth study. Of these only three were true interaction faults, while the remainder were faults contained within code for an optional feature or else faults in the configuration-realizing code. Although all of these non-interaction faults could have been found through better feature-level testing, we observed that testing under different configurations traverses a richer subset of the system's behaviors and may complement feature-level testing.

In future work we plan to build a mutation testing tool specifically for seeding interaction faults. We also intend to extend this study to a larger set of subjects and to evaluate the potential for using configuration testing techniques to improve feature-level testing in configurable software.

## Acknowledgements

We would like to thank Wayne Motycka for helping validate data. This work is supported in part by the National Science Foundation through award CFDA#47.076 and CCF-0747009 and by the Air Force Office of Scientific Research through award FA9550-09-1-0129. 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 or AFOSR.

## References

[1] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Intl. Symp. on SW Testing and Analysis*, Jul. 2008, pp. 75–85.

[2] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. on SW Eng*, vol. 30, no. 6, pp. 418–421, 2004.

[3] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Trans. on SW Eng*, vol. 31, no. 1, pp. 20–34, Jan. 2006.

[4] 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 Trans. on SW Eng*, vol. 34, no. 5, pp. 633–650, 2008.

[5] B. Robinson and L. White, "Testing of user-configurable software systems using firewalls," in *Intl. Symp. on SW Rel. Eng.*, Nov. 2008, pp. 177–186.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Trans. on SW Eng*, vol. 23, no. 7, pp. 437–444, 1997.

[7] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Intl. Conf. on SW Eng.*, 1999, pp. 285–294.

[8] Y.-J. Lin and M. Jazayeri, "Managing feature interactions in telecommunications software systems - guest editorial," *IEEE Trans. on SW Eng*, vol. 24, no. 10, pp. 777 –778, Oct. 1998.

[9] S. Nejati, M. Sabetzadeh, M. Chechik, S. Uchitel, and P. Zave, "Towards compositional synthesis of evolving systems," in *Intl. Symp. on Found. of SW Eng.*, 2008, pp. 285–296.

[10] A. Classen, P. Heymans, and P.-Y. Schobbens, "What's in a feature: a requirements engineering perspective," in *Theory and Practice of SW, Intl. Conf. on Fund. Approaches to SW Eng.*, 2008, pp. 16–30.

[11] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *Intl. Conf. on SW Eng.*, may 2010, pp. 445–454.

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Secur. Comput.*, vol. 1, pp. 11–33, January 2004.

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

[14] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Intl. Conf. on SW Eng.*, 2010, pp. 105–114.

[15] M. Nita and D. Notkin, "White-box approaches for improved testing and analysis of configurable software systems," in *SW Eng. - Comp. Vol, Intl. Conf. on*, May 2009, pp. 307 –310.

[16] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *SW Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.

[17] P. Zave, "Feature interactions and formal specifications in telecommunications," *Computer*, vol. 26, no. 8, pp. 20 –28, 30, Aug. 1993.

[18] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Trans. on SW Eng*, vol. 27, pp. 228–247, March 2001.

[19] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at system and functional levels," in *Intl. Workshop on Mutation Analysis*, apr 2010, pp. 110–119.

[20] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. on SW Eng*, vol. 32, no. 8, pp. 608–624, 2006.