Configurations Everywhere: Implications for Testing and Debugging in Practice

Dongpu Jin
Dept. of Comp. Sci. & Eng.
Univ. of Nebraska-Lincoln
Lincoln, NE 68588, USA
djin@cse.unl.edu

Xiao Qu
Industrial Software Systems
ABB Corporate Research
Raleigh, NC, 27606, USA
xiao.qu@us.abb.com

Myra B. Cohen
Dept. of Comp. Sci. & Eng.
Univ. of Nebraska-Lincoln
Lincoln, NE 68588, USA
myra@cse.unl.edu

Brian Robinson
ABB Inc.
Raleigh, NC, 27606, USA
brian.p.robinson@us.abb.com

ABSTRACT

Many industrial systems are highly-configurable, complicating the testing and debugging process. While researchers have developed techniques to statically extract, quantify and manipulate the valid system configurations, we conjecture that many of these techniques will fail in practice. In this paper we analyze a highly-configurable industrial application and two open source applications in order to quantify the true challenges that configurability creates for software testing and debugging. We find that (1) all three applications consist of multiple programming languages, hence static analyses need to cross programming language barriers to work, (2) there are many access points and methods to modify configurations, implying that practitioners need configuration traceability and should gather and merge metadata from more than one source and (3) the configuration state of an application on failure cannot be reliably determined by reading persistent data; a runtime memory dump or other heuristics must be used for accurate debugging. We conclude with a roadmap and lessons learned to help practitioners better handle configurability now, and that may lead to new configuration-aware testing and debugging techniques in the future.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

Configurable Systems, Testing, Debugging

1. INTRODUCTION

Many software systems are *highly-configurable*, allowing the user to customize an individual instance of the program while retaining a core set of functionality. This customizability provides benefit to the end-user, however, it also introduces many challenges during testing and/or debugging, because configurability complicates the process of finding and/or reproducing the failure. Research has shown that different instances of a highly-configurable system will behave differently while running under the same set of test cases [6, 21, 28]. For instance, in the work of Qu et al. [21], as many as 80% of the faults had the potential to go undetected if tested under certain configurations. Therefore, configuration-aware testing techniques have been proposed, to systematically explore the configuration space [21, 28]. But all of this work assumes that the configuration model is known (or is easily extracted). During debugging, this is also important. Knowing the exact configuration instance that a user was in when the failure occurred can help with reproducibility. Bettenburg et al. [1] found that there is a strong mismatch in bug reports between what developers need to reproduce and fix a bug, and that which is provided by users. Other studies have also shown that bug reports lack information needed for bug reproduction [2]. There has research aimed at reproducing field failures [3, 14], but this line of work does not explicitly consider the configuration at the time of failure.

Given the complexity of today's software systems, determining the configuration space may not be a trivial task. For instance, in the industrial system studied in Qu et al. [20], they reported that there are more than 500 configuration options that their users can modify. Firefox, the open source web browser has over 1,900 configuration options available to a user. The space of possible unique configurations grows exponentially with the number of configuration options (or preferences), therefore we can only evaluate a representative sample of all possible configurations.

As we work more and more with highly-configurable systems in practice, we have discovered common issues that arise which make available configuration-aware techniques insufficient. For instance, in our industrial systems there is usually no single document that describes the complete set of possible configuration options. We can examine external preference files, but we find that there may be multiple files, and they still tell only a partial story because there are *hidden* (but valid) preferences found only in the source code. We can try to use an analysis technique such as those proposed by Rabkin et al. [23,24] to reverse engineer a complete mapping of our configuration space, but many applications are written in multiple programming languages such as C++, Java and JavaScript, and often use aliasing to refer to preference names, neither of which are supported by existing techniques. Finally, if we assume that we can

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '14, May 31-June 7, 2014, Hyderabad, India

Copyright 2014 ACM 978-1-4503-2768-8/14/05 ...$15.00.

somehow obtain the ground truth model of the configuration space, then in order to manipulate the configurations for testing and debugging, we need mechanisms to automate this process, as well as ways to capture *which* configuration was active during a failure. Again, we have learned that the complexity of real software makes this difficult – configurations can be modified and viewed from multiple locations, and are found in both dynamic and static structures. Finally, we have discovered that it is possible for the static structures to be out of synchronization with the dynamic ones at the time of failure.

In this paper we attempt to uncover and quantify the extent to which these problems exist on an industrial scale. We empirically examine several large highly-configurable applications to understand the implications for testing and debugging in practice. We study one industrial application and two widely used open source applications. We quantify the size of the configuration space and evaluate where and if the ground truth for the configuration model exists. We also examine how a user, tester or maintenance engineer can manipulate the configuration options. Finally, we examine the runtime factors involved in capturing the current configuration space. Our study shows, somewhat surprisingly, that both the industrial and open source applications have elements of configurability in common, which leads to a set of lessons learned and a roadmap for developing configuration-aware testing and debugging tools. We see this study as a way to share with practitioners the issues configurability brings, and a springboard to accurate and usable configuration-aware testing and debugging techniques.

The contributions of this work are:

1. An abstraction of the general structure of configuration manipulation in modern software systems
2. A case study that quantifies the complexity of three modern highly configurable software systems
3. A set of lessons learned that will help practitioners to better understand and control configuration instances for software engineering tasks such as testing and debugging

The rest of this paper is structured as follows. In the next section we present some motivating examples motivating our study. In Section 3 we present our case study. Results are then given in Section 4, followed by lessons learned in Section 5. We highlight related work in Section 6 and conclude and present future directions in Section 7.

2. MOTIVATION

A *configurable system* is a software system with a core set of functionality and a set of variable features which are defined by a set of *configuration options* (or *preferences*).[1] Changes to the *value* of a preference changes the program's behavior in some way. For instance, Firefox, a popular web browser, is a highly-configurable system and one that we use to motivate some of the problems that we have encountered. In Firefox, an example configuration option that can be set via the option menu is called *Warn me when closing multiple tabs*. This is a Boolean configuration with two values, {*true, false*}. Its default value is set to *true* which means that if you try to close a window when multiple tabs are open, you will get a warning asking if you want to close all of the tabs. If you uncheck this on the menu (set it to *false*) it will prevent a warning from being produced and immediately close the window. The actual preference name for this (found in the preference file) is called *browser.tabs.warnOnClose*. There is another closely related preference in the preference files called *browser.tabs.warnOnCloseOtherTabs* which is set to *true* by default, but has no menu counterpart. When testing the system, or when a failure occurs, we need to have information about the values that were selected for each of these configuration options, something that may not be obvious by examining just the menu alone.

[1]In the remainder of this paper, we use configuration option and user preference interchangeably.

We assume an idealized use case for testing and debugging as shown in Figure 1. In this scenario we have three entities that interact with the configurable system. The end-user can modify configurations and will send bug reports to (and possibly read reports from) customer support. As can be seen in the figure, he or she may use the menu, or they can directly write to configuration files. A set of configuration-aware techniques and tools sit between the application and the tester and maintenance engineer, which feed information about configurations back to the bug reporting/customer support system. The challenge is to enable these configuration-aware techniques. We have identified three important requirements. We need

1. **a Model** of the possible configuration space. In order to sample the configuration space for testing or debugging, the configuration model needs to be known,
2. **to Know the Mapping** of the configuration space to programmatic elements. This is required in order to understand the impact a configurable item can have, and to automate the modification of configurations for testing and bug reproduction, and
3. **an Accurate Configuration Snapshot** to provide the full state of the application when a bug is encountered.

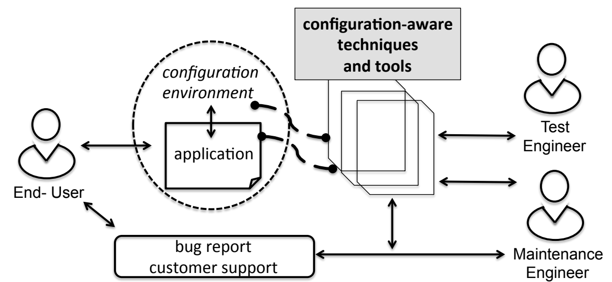

Figure 1: Configuration-aware testing and debugging: expected use case

We examine each of these requirements in relation to the existing work. Configuration-aware testing techniques [5, 10, 21] propose various methods to sample and prioritize the configuration space for testing, but all of this work assumes that the **c**onfiguration model is known (or is somehow extracted from the code). Based on our informal examination of systems like Firefox, we do not believe that this can be easily achieved. First, we have discovered that the configuration control is not found within a single location of the code or in specific external files. In fact, most of the systems

we have studied have a multi-tiered layout of how configurations are defined and accessed and this can be done both offline and at run time. Figure 2 shows a schema that represents all of the systems we have studied. First, there is a *static view* of the system (labeled #1). This includes any existing user manuals, web pages, etc. that contain documentation on the possible configuration options and their values. This often is incomplete or out of date (see our discussion of hidden or dead preferences in Section 5). The second static element is the source code itself. This contains the *ground truth*, but source code may not be available to everyone who wants and needs to understand the configuration model. Moreover, as we shall see, using this to extract the full configuration space is non-trivial.

When controlling what configurations are set, there are usually external mechanisms (#2 in Figure 2) such as preference files or databases. These can often be accessed independently of the program (even while it is running) and therefore may or may not contain the current state of the configurations. We have also seen that these may not contain the ground truth of the configuration space.

Finally, as is shown in #3, there are usually some runtime access mechanisms that connect to the internal data structures (or database). For instance, most programs have a menu system that allows the user to set preferences, but in the systems we have studied this accounts for only a subset of the full set of configurations. Other specialty tools exist such as the `about:config` mechanism of Firefox, that allows one to pull up a web page where configurations can be modified dynamically. Again, these may not show the complete set of configuration options that are available. There may also be an API to allow programmatic access to an internal memory structure (such as the hash table in Firefox). This should be the ground truth of what preferences are set at any point in time, but it will not contain the hidden preferences.

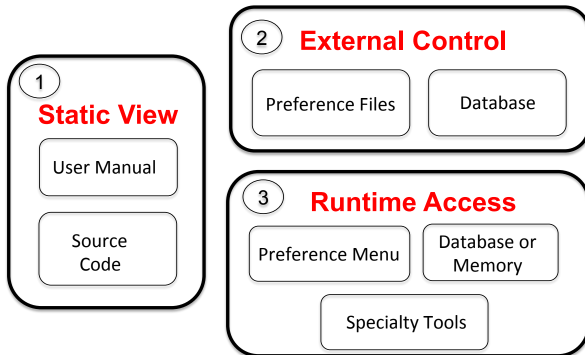

Figure 2: General view of configuration layers

Suppose instead of using the menus or preference files, we want to extract the preferences from the code itself, which also helps to build a **mapping** between the configuration space and code. Rabkin et al. [23,24] presented techniques to statically analyze Java programs with JChord. Upon studying their work in more detail, we find that it does not directly apply to a system like Firefox. First, it assumes a single programming language (Java); second, they assume all of the preference manipulation code exists as (`name, value`) pairs and is found in a single class; and finally, they assume that configuration manipulation methods start with *get* or *set*.

As shall see, these assumptions do not hold for any of the applications we studied. For instance there are instances in Firefox where the preference code includes JavaScript and other languages such as the markup language XUL. We see instances where the Javascript API is able to query and update a preference, however, it uses the XUL code as a reference to the given preference name (binding it to a user interface element). We also see preference code that is not using the (`name, value`) pair mechanism but instead uses references, macros, or member fields to refer to the preference name. Another issue that we have encountered is that the API method names of Firefox do not always start with *get* or *set*. We need more different techniques if we plan to extract all of these configuration options from the code.

Finally, if we are concerned with knowing the current state of the configuration space at some point in time, we need a technique that captures an **accurate configuration snapshot** at runtime. Indeed it may not be straightforward to get this information from the system. In some of the applications we have studied (Firefox and LibreOffice), when the user modifies a preference value dynamically through the option menu, the change is reflected immediately in the dynamic memory and preference files. However, in our industrial application, the change made by the user will be stored temporarily and the new preference will take place on the next startup. Therefore the running configuration and the one reflected in the persistent memory after the application closes may be inconsistent.

Faced with the complexity that we have described informally so far, we want to quantify how often we see these problems with the aim of developing a generic model of how modern highly-configurable software is structured and manipulated. We also want to know if there is a ground truth for the configuration model and dynamic configuration states in modern configurable systems. We next present a case study that we have developed for this purpose.

3. CASE STUDY

Our study has two main objectives. First, we want to quantify the complexity of the configuration space and what mechanisms are used to define and manipulate this space. Second, we want to understand what are the challenges that we will face as we develop configuration-aware testing and debugging techniques. To address these issues we will center our study around answering the following research questions.

RQ1: *What is the complexity of the configuration space in modern configurable software systems?*

RQ2: *How are configuration options structured, changed and accessed by the user in these systems?*

RQ3: *Are the selected configuration options synchronized between the different parts of the system and throughout the lifecycle of program execution?*

3.1 Software Subjects Studied

We have selected three different software systems to study. The first is a large real-time embedded software system developed at ABB (called ABB_c hereafter). ABB_c has approximately 10 Million lines of code, is highly-configurable, and has more than 58 modules; each module defines a subsystem that implements a different set of functionality of the system. The second subject, Firefox [19], is an open source web browser which works on multiple operating systems and has over 300 Million users worldwide and over 9.6 Million

lines of code. The third subject is LibreOffice. It is an open source office productivity suite consisting of a word processor, spreadsheet application, presentation tool, drawing application, math formula tool and database [9]. LibreOffice has 6.8 Million lines of code and 25 Million users worldwide estimated by The Document Foundation in 2011 [27]. We believe that these subjects are different from each other in functionality and in development style, yet they are all large, highly-configurable, and used in a production environment by a large pool of users.

3.2 Study Design

To answer our research questions, we collect configuration information from both a static and dynamic perspective on each system. We manually study all artifacts that are publicly available to users, including documents (e.g., user manuals and online help pages), software option menus on the user interface, preference files and source code. We also utilize tools or APIs that have been provided to programmatically manipulate internal data structures that hold configuration information. For ABB_c we have a user manual that is written for those who will modify and change preference files. In addition, we have asked questions of developers to confirm our assumptions. In Firefox we utilize the source code, examine the internal dynamic data structures via an API call when the application is running, as well as study the `about:config` page (a utility for modifying configurations). We also study the `Options` menu, the SQLite database that holds page specific preferences, and online documentation. For LibreOffice, with the help of online documentation, we study the preference files and use an API to connect to the dynamic data structures when the program is running. To answer RQ1, we calculate the ABB_c configuration space based on the user manual and we calculate the configuration space for Firefox and LibreOffice through a program we wrote to query the dynamic data structures at runtime.

When we collect the configuration information, we make some assumptions. First, constraints between options are ignored. We realize that this might over approximate the configuration space slightly, but extracting the exact configurations options may not be feasible without in-depth knowledge of each system. Second, the plug-ins (add-ons) are not included in our calculations. In Firefox and LibreOffice, we build clean versions of the system from source code for study. Any default plug-ins that come with those will have their configuration options included, however no additional plug-ins are enabled. To calculate the number of values associated with an option, we have detailed information for many of the configuration options in the ABB_c manual. However, when they are not available, and for Firefox and LibreOffice, we use a set of rules to come up with a small set of categories. For Boolean configuration options we use *true* or *false*. For integers we use a 'default value', a 'non-default legal value' and an 'illegal value', resulting in 3 values. For strings we use 'no string', an 'empty string' and a 'legal string', again resulting in 3 values. In ABB_c we have some strings with constraints. For these we use 4 values by adding an 'illegal string'. This partitioning may underestimate the true configuration space, (it is a conservative model), but it is consistent with prior work [5].

For RQ2 and RQ3 we analyze the systems further and experiment with the various ways that one can modify configurations when the system is not running. We also analyze what happens if configurations are modified while the programs are running as well as what occurs with the changed configuration options during startup and shutdown. We examine some of the preference setter code and also look for hidden preferences that may not have been exposed earlier. We look at both menu access as well as file access. We also use the specialized tools such as the `about:config` to interface with Firefox and the ABB tools (denoted as ABB_a and ABB_b) to interface with ABB_c.

3.3 Threats to Validity

As with any study there are threats to validity which we document here. First, we have only studied three software systems. While we believe they are different enough (one is an industry application while two are open source applications with different sets of developers) we can not be sure that our results will generalize to all configurable applications. Our second main threat is that we are not developers of these systems so we have relied on the documentation and code to extract the information that we need. With ABB_c we were able to confirm our questions with developers. In the Firefox and LibreOffice environment we do not have this as a source of validation. But we used third party APIs that are commonly used to interact with the configuration environments and made an effort to validate our result internally. We have made the tools we used to query Firefox and LibreOffice available online as well as the artifacts that we have obtained to reduce this threat. Finally, we could have measured different elements for this study, but feel that the set of metrics we collected supports our research questions.

4. RESULTS

We now present our results for each of the three research questions. Supplemental data for the open source applications (and our tools for gathering the data) can be found on our associated website (see `http://cse.unl.edu/~myra/artifacts/SEIP-2014/`).

4.1 RQ1 Configuration Complexity

To answer RQ1, we turn to Tables 1, 2 and 3. Table 1 provides the basic statistics for our applications. It first shows the operating system and versions of the two open source applications. We then list the primary languages that are used in each application. We show all languages that make up at least 1% of the code. We leave out markup languages such as XML or XUL. All three applications consist of at least two languages. Firefox has the most with C++, C, JavaScript, Python, Assembly and some shell script. LibreOffice has both C++ and Java. ABB_c has a mixture of three languages, C++, C and C#. We also list the number of preference files that are used to store the current set of preferences and that are read at startup. As we see, this ranges from 6 files in ABB_c to 193 in LibreOffice (there are six preference files in ABB_c, but we were unable to access one of them, so all of the computation that follows uses only five files). Finally, we list the total numbers of unique preferences that we counted in each of these applications. This ranges from 524 in ABB_c to 36,322 in LibreOffice.

We next look at Tables 2 and 3. We show a breakdown of the configuration options by the data types and number of values associated with each type. Table 2 has data for ABB_c and Firefox. As we can see, we have only three types

Table 1: Quantifying number of preference files and preferences of ABB_c, Firefox and LibreOffice

	ABB_c	Firefox	LibreOffice
Operating System	Embedded System	Ubuntu 12.04	Ubuntu 12.04
Version	-	Mozilla Firefox 27.0a1	LibreOffice 4.0
LOC (M)	10.0	9.6	6.8
Primary Languages	C++(3.7%), C(29.6%),C#(8%)	C++(41%),C(21%), JavaScript(16%),Java(3.1%), Python(2.7%), Assembly(1.2%), Shell script(1%)	C++(82%), Java(6%)
Total Pref Files	6	11	193
Total Prefs	524	1957	36322

Table 2: Categorization of configuration space for ABB_c and Firefox. The total number of preferences are shown as c^n where c is the cardinality of the preference (number of values) and n is the number of times we have this cardinality). We have combined like cardinalities together therefore the total *boolean* values for example may include some from the *others* category.

Types	ABB_c	Firefox
Boolean (2)	92	846
Integer (3)	271	517
String (3)	27	594
String with condition (4)	110	–
Others	24	–
Total	$2^{96}3^{303}4^{114}6^{4}7^{1}8^{3}9^{1}16^{1}18^{1}$	$2^{846}3^{1111}$

Table 3: Categorization of the configuration space for LibreOffice broken down by module

Types	Writer	Calc	Impress	Draw	Math	Database	Others	Total
Boolean (2)	201	58	69	44	77	44	3940	4433
Integer (3)	157	43	26	22	110	15	5087	5460
Others	298	70	32	3	141	167	25718	26429
Total	$2^{201}3^{455}$	$2^{58}3^{113}$	$2^{69}3^{58}$	$2^{44}3^{25}$	$2^{77}3^{251}$	$2^{44}3^{182}$	$2^{3940}3^{30805}$	$2^{4433}3^{31889}$

in Firefox resulting in 846 boolean options and 1,111 options of either integer or string, each with three values. The total configuration space is equal to $2^{846} \times 3^{1111}$. ABB_c has a variety of cardinalities for its configuration options. We have a more exact model due to better documentation. Our total configuration space for this application is 6.46×10^{259}.

Finally we look at Table 3 which shows the configuration options in LibreOffice broken down by individual modules within the suite of tools. This is based on the hierarchical path used to display the configuration option name. For instance all of the preferences under `Writer` have the prefix `org.openoffice.Office.Writer`. We do not believe that all 36,322 would be used together in any test or debug model. Instead one would test an application such as `Writer` individually. Although we can identify which preferences belong to specific applications such as `Writer` or `Calc`, there are some categories such as `UI` which may be shared among applications. These all fall into the `Others` category. The complete categorizations are contained on our website.

ABB_c has preference files that contain additional information not found in the open source applications. This is because it is an embedded system with configuration options that can be customized for different drivers or ports. The number of devices and ports is open ended. The two additional pieces of information in these preference files are *category* and *instance*. Certain preferences are grouped into a category, and for each category we have one or more instances that consist of the same set of preferences. Each category may contain multiple instances, therefore one preference can appear multiple times. To understand this better, we can consider a situation where each instance is associated with a specific hardware or virtual device. Some devices are in the same category, thus have the same set of preferences, however the device that is being controlled differs.

An example of a snippet of the ABB_c preference file is illustrated in Figure 3 (the names are changed for proprietary purposes). There are five options in this figure (bold fonts): **Name** (string), **Count** (integer), **Unit** (string), **Length** (integer), and **Status** (boolean). **Name** and **Count** are grouped under `CATEGORY A`, while **Unit**, **Length**, and **Status** are grouped under `CATEGORY B`. There are three instances in `CATEGORY A`: in the first instance (line 3), the **Name** is assigned with value `x` and **Count** is assigned with *2*; in the second instance (line 4), the **Name** is assigned with `y` and **Count** is assigned with *5*; in the third instance (line 5), the **Name** is `z` and **Count** is the default value.[2] Similarly, there is one instance in `CATEGORY B` (line 8): the option **Unit** is assigned with *X*, the **Length** is assigned with *10*, and the **Status** is assigned with *ON*.

Table 4 shows the number of configuration options grouped by *categories* and the number of categories for each preference file. In this paper, when we compute the configuration space shown in Table 2, we made a conservative assumption that all options will appear a single time (regardless of instances), to make it in consistant with other systems.

[2]The ABB_c user manual states that "if the option is assigned the default value, then it will not be listed in the configuration file" and this is why the third instance only has one option explicitly written.

1. #

2. CATEGORY A:

3. -**Name** “x" -**Count** “2"

4. -**Name** "y" -**Count** " 5"

5. -**Name** "z"

6. #

7. *CATEGORY B*:

8. -**Unit**" X" -**Length** "10" -**status** “ON"

Figure 3: Example of ABB_c preference file

Table 4: Number of options grouped by categories in ABB_c

Preference Files	Number of Categories	Number of Options
File 1	3	26
File 2	11	50
File 3	10	78
File 4	7	22
File 5	39	348
Total	70	524

4.2 RQ2 Configuration Access

We begin answering RQ2 by examining the structure of one of our open source systems, Firefox. Figure 4 shows this schematically. In this figure there are a number of preference files (both user and default) that contain values for specific preferences. During the application startup, the default configuration options are read (there are 1932 of them), and after that, the user preferences are read (there are 50 of them initially). These are read by the preference modules. The user can modify these on disk directly if they understand the format. The next time the application opens, these files will be read (assuming that they have not been overwritten in the meantime – see RQ3 for a discussion of that mechanism) and the preferences will be activated. The user can also open Firefox and use the `about:config` webpage to control (or look at) the preferences. If a user modifies a preference in the `about:config` it will be written to the user preference file and be set via the preference modules in the code. Additionally the user can go through the options menu. This contains only a subset of the full set of possible options, only 126 out of the 1957 (calculated in Table 1). We do not quantify (or discuss) the Add-on configuration options in this paper, but these are also manipulated through a menu. Finally, there is an SQLite database which contains page-specific option settings for the browser (e.g. if a user zooms in on a particular website, this information will be stored for the next time they open that site).

The preference modules are accessible through a set of preference APIs. The APIs are used to interface with a dynamic hash table which contains all active configurations when an application is running. There is a 1 to 1 mapping of the preference files to the hash table, but an N to 1 mapping of the menu items. These are used as variables in the code and several names may map to the same individual option in memory. Finally the code itself (program modules) contain the ground truth for the configuration space. We have discovered several options in the code that are *hidden*. These are options without default values that can be set if a user

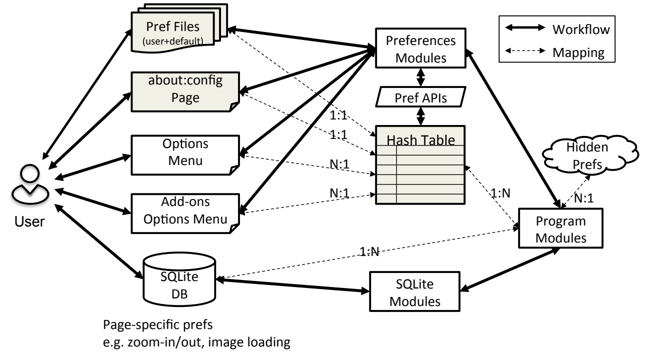

Figure 4: Firefox configuration structural diagram

knows about them, but which do not appear in our results for RQ1 since they are not in the hash table or preference files unless explicitly set by the user.

We have analyzed the user interface (UI) source code of the Firefox option menu and retrieved 126 preferences that are bound to the option menu UI elements. Listing 1 shows an example of binding the preference browser.startup.page, (which specifies the start-up page Firefox opens), to a drop-down menu list in the option menu. Thus only 6.4% of the total preferences exist in the option menu in Firefox.

We note that both the ABB_c and LibreOffice systems have similar structures, therefore, we do not show them all here, but an extraction of the general structure is illustrated in Figure 2 and introduced in Section 2.

We next investigate how configuration values are read in the code. First, we take a look at the APIs used to access the configurations in the code. In Firefox, the return value is almost always passed by reference. For example, the signature of a boolean preference access functions from the source file *prefapi.h* under */modules/libpref/src* is shown in Listing 2. As we can see, the configuration option value *return_val* is passed as a pointer in the formal parameter list. The function returning value (i.e., *nresult*) is just an binary indicator of whether the actions defined in this function succeed or fail. This prevents us from using the techniques developed by Rabkin et al. [23, 24] because the preference type cannot be inferred by tracking return value types.

```
// main.xul
<preference id="browser.startup.page"
  name="browser.startup.page" type="int"/>
 ...
<menulist id="browserStartupPage"
     preference="browser.startup.page">
  <menupopup>
     <menuitem label="&startupHomePage.label;"
          value="1" id="browserStartupHomePage"/>
     <menuitem label="&startupBlankPage.label;"
          value="0" id="browserStartupBlank"/>
     <menuitem label="&startupLastSession.label;"
          value="3" id="browserStartupLastSession"/>
  </menupopup>
</menulist>
//main.js
let startupPref =
     document.getElementById("browser.startup.page");
 ...
startupPref.updateElements();
```

Listing 1: Setting Firefox preferences using XUL

```
nsresult PREF_GetBoolPref(const char *pref, bool
    *return_val, bool get_default);
```

Listing 2: Return value is passed by reference

Second, the preferences are accessed via multiple programming languages. The Listing 1 and 3 show two examples of the Firefox source code interfacing with the preference system via XUL and JavaScript respectively. The JavaScript performs most of the manipulation, but the XUL code interfaces and dereferences the preference name.

```
// nsBrowserContentHandler.js
var choice = prefb.getIntPref("browser.startup.page");
// nsBrowserGlue.js
Services.prefs.setIntPref("browser.startup.page", 3);
```

Listing 3: Setting Firefox preferences using JavasScript

Third, the preference name can be in various forms when passing to preference APIs. The name of the preference is usually passed as the first parameter to the preference APIs. Listing 4 shows a few examples of passing the preference name as a string, a variable, an object macro, a function macro, or a class member.

```
// String
rv = mPrefBranch->GetBoolPref(
    "autoadmin.append_emailaddr", &appendMail);
// Variable
prefBranch->GetIntPref(
    kCookiesLifetimeBehavior, &lifetimeBehavior);
// Object macro
rv = branch->GetIntPref(
    DISK_CACHE_CAPACITY_PREF, &capacity);
// Function macro
rv = prefs->GetIntPref(
    HTTP_PREF("connection-retry-timeout"), &val);
// Class member
rv = prefBranch->GetBoolPref(
    externalProtocolPref.get(), &externalProtocol);
```

Listing 4: Different types of API preference name parameters

Finally, we show examples of *hidden preferences*. In the `String` example in Listing 4, the preference `autoadmin.append_emailaddr` appears in the source code, but it does not exist in any preference files unless added by the user.

Preferences shown in Listing 5 are some other examples of hidden preferences from Firefox source code. Our configuration space analysis (RQ1) misses these preferences. We do not know how many exist in Firefox.

```
pref.browser.homepage.disable_button.bookmark_page
pref.browser.homepage.disable_button.current_page
pref.browser.homepage.disable_button.restore_default
```

Listing 5: Hidden preferences

We also investigate how configuration values are read in the code in ABB_c. First, there is a configuration manager class (written in C) that reads the values at different levels: it may read values of a single preference, it may read a single instance that contains a couple of preferences, or it may read all instances that under the same configuration category. Just like in Firefox, all these values are passed by reference. Second, the name of the preferences can be in various forms, such as string, variable, and macro. Finally, there are several configuration options that are accessed in the code but not in the document (hidden preferences) and there are also some configuration options that are in the document but are never read in the code (dead preferences).

Table 5: Number of configurations accessible at different layers

System		Static View		Extern Control	
	Tab. 1	Code	Manual	Files	Menu
ABB_c	**524**	$428 + 166$	**524**	< 524	< 524
Firefox	**1957**	> 1957	NA	> 1957	126

Table 5 summarizes the number of configuration options that are accessible at different layers (defined in Figure 2). The first column (Table 1) shows the values we obtained for RQ1. The last column (Menu), is used to represent configuration control via menu in Firefox and via ABB_a and ABB_b in ABB_c. For ABB_c there are $(428 + 166 = 594)$ options accessed in code. 428 options are also described in the manual, but 166 options only appear in code (hidden preferences), and $(524 - 428 = 96)$ options only appear in the document (dead preferences). This shows that the document is not updated accordingly as the code is changed, although the document is a very important artifact that tightly connects the system with customers. We do not have accurate numbers of the preferences accessible by external control elements, but quote the manual which says "if the option is assigned the default value, then it will not be listed in the preference file."; there are also preferences not in ABB_a or ABB_b given that "some configurations have to be changed in preference files".

4.3 RQ3 Configuration Synchronization

To answer RQ3, we map the lifecycle of a running application to understand when and where its configurations are synchronized between its layers. We model three distinct phases, *startup*, *runtime*, and *shutdown*. Figure 5 shows the behaviors of Firefox and LibreOffice, and Figure 6 shows the behaviors of ABB_c. The numbers on the leftmost side specifies the number of preference files in different groups of files. Solid arrows represent direct connections, while dashed arrows indicate the need for a mapping/traceability.

In all three systems at startup, the configurations are read from persistent storage (configuration files) and loaded into memory. There is a specific order in which these are loaded. If the same configuration options are repeated, set to different values, the last one read will be the one which holds. While the applications are running, a user can modify the configuration files directly. This is not immediately reflected in the dynamic memory. If a failure occurs at this point the persistent memory is out of sync with the dynamic. In all three systems the user can also dynamically modify the configurations while the application is running. In Firefox and LibreOffice these will take effect immediately and be written back to the preference files. In ABB_c the dynamic memory is not updated. The changed configurations are held in temporary memory and take effect at the next startup.

On shutdown, in Firefox and LibreOffice the dynamic memory overwrites the current preference files before the

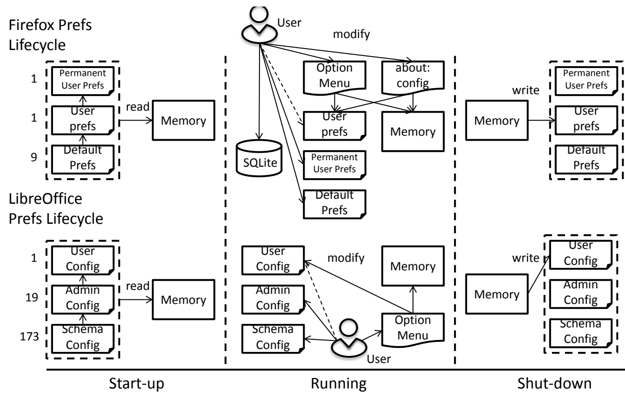

Figure 5: Firefox and LibreOffice lifecycle diagram

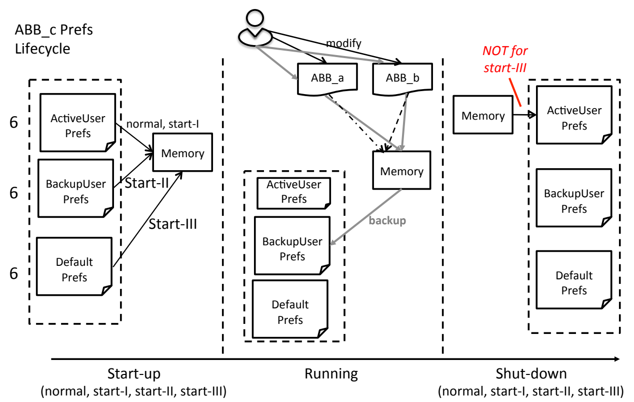

Figure 6: ABB_c lifecycle diagram

application closes. In Firefox the user preference file is overwritten, but the default ones are not. This means that if a user modified the user preference files during runtime, those changes will never be seen (not even on the next startup). However, if they modified other preference files they will appear on next startup.

ABB_c has a more complicated "restart" behavior described next. When the system is restarted normally (denoted as start-I): the current system will be stopped. All system preferences will be saved. Restarting this way will activate any configuration changes. A second option is to restart and select another configuration (start-II). In this case the current system will be stopped. All system preferences will be saved, so that the system state can be resumed later. The last restart is to restart and return to default settings (start-III). After restart, the system state will be resumed but any changes done to system preferences will be lost. Instead, system preferences are read from the originally installed system on delivery.

Furthermore in ABB_c there are three sets of preferences: active (loaded by default), backup, and default. During startup, instead of loading different sets of preferences in order (as happens in Firefox and LibreOffice), the system only loads one set of preferences into memory, based on the type of restart. During normal start and start-I, the active preferences are loaded, during start-II, a selected set of previous backup preferences are loaded, and during start-III, the factory default preferences are loaded. During run time, the users can make configuration changes in preference files directly, or through ABB_a or ABB_b, but changes will not take effect until a restart. The changes will be stored temporarily in a memory different from the active preferences. Users can also save the currently active preferences as a backup. Finally, all changes made at the runtime will be written back into the active preference files when the system is normally shutdown or restarted in I or II. Note that if the users select a start-III, all changes will be lost.

5. LESSONS LEARNED

In this section we summarize the implications and lessons learned from our study. The first two lessons learned are geared towards practitioners since they reflect the state-of-the-art. The last two provide a roadmap for researchers who plan to develop new tools and techniques for configuration-aware testing and debugging.

1. Configuration Modeling Should Merge Multiple Layers

We return to our first question of how one can model the full configuration space when performing testing and debugging. Although the application code is the *ground truth*, the maintenance engineers may not always have access to code. If instead we use the user manuals/documentation, we most certainly miss out on some configurations. Moreover, in the applications studied, the menu on the user interface contained only a small subset of the configuration options. While these might contain the most widely used preferences, they do not provide a true indication of the real configurability of a system. Finally, we can use the persistent configuration preference files, but we must first understand how (in what order) and when these are activated in the dynamic system. Two issues that have arisen during our analysis are those of *hidden preferences* and *dead preferences*. These constitute a small part of the configuration space model, but one should be aware of their potential existence. Given the results of our study, we believe that to obtain an accurate model of the configuration space one should consider and merge multiple artifacts which includes preference files, menus and documentation. Additionally, since documentation is the primary artifact a user would read, it should be updated as the design and code changes in a timely manner, particularly when it comes to system testing or other configuration related tasks.

2. Configuration Traceability is a Necessity

Given the variety of places that configurations are accessed and mentioned, it seems that the task of simply setting a configuration option requires deep knowledge of the application. If we return to our example, Firefox, one needs to know the mapping of menu names to preference variable names to modify them automatically. Furthermore we have seen (both in Firefox and ABB_c), a many-to-one mapping of variables in the code and preference files and dynamic memory. Providing traceability mapping between elements of the configuration manipulation mechanism are essential to making configuration-aware techniques work.

3. Analysis Tools Need to Cross the Programming Language Barrier

As we have seen, the current state of research in analysis for extracting configurations from code expects a sin-

gle programming language and single class files where the configuration information code (such as setting and getting configuration) lies. Yet this is not realistic for the large scale subjects that we have studied. Our configuration options are manipulated and referenced across programming language barriers and in multiple modules. We need, therefore, new analysis techniques that cross these boundaries, can handle aliasing, and that use additional heuristics to identify the actual getter and setter code.

4. Configuration State Capture or Approximation Techniques are Needed

As we argued at the start of this paper, we need a way to capture the active configuration when the system fails so that we can reproduce and debug the failing test case. Each of the three systems we studied, allows the user or maintenance engineer to modify the configurations both externally or internally during runtime. While our open source applications update the memory and files immediately, in our industrial application, the configuration is not activated until possibly startup (with the exact behavior dependent on the type of reboot selected). Even if we understand how the configuration manipulation works, there is the possibility of race conditions in all of the applications, depending on the exact timing of the configuration modification and failure. It is also possible to make changes to external files for modifications at startup, yet these may be overwritten during a normal shutdown. In order to extract the ground truth of the *configuration at failure*, monitors are needed that capture this information. But these may incur overhead and cause concerns for privacy. Alternatively, we know that the persistent memory contains a large portion of the correct configuration space, so algorithms that work from this point and search close by may be useful for reproducibility. Research has shown that failures tend to have *feature locality* [11], so it is possible we can leverage some of those ideas for this work.

6. RELATED WORK

We provide a short overview of several areas of research that are closely related to this work. The role of software users and essential information in bug-fixing has been emphasized in several studies [1, 2, 25, 31]. Bettenburg et al. [1] found that there is usually a strong mismatch in bug reports between what developers need to reproduce and fix a bug and what is provided by users. Herbold et al. [13] developed a tool to capture usage logs for replaying bugs. Other work tries to reproduce field failures [3, 14], however the focus is on using the call graph. None of this work tries to capture the software configuration used during the failure.

Several researchers have been focusing on extracting configuration options from code. Rabkin et al. [23, 24] propose a method to statically detect system configurations, but as already mentioned this analysis works on a single language (Java) and assumes that all configurations are contained in a single class. Yin et al. [29] conducted empirical studies to understand the configuration errors in commercial and open source systems. Zhang et al. [30] have proposed a technique to diagnose crashing and non-crashing errors related to software misconfigurations. Again their tool only works on a single language (Java) and the configurations they study are simple. We look at more complex configuration spaces with multiple languages and multiple preference layers, etc.

From a traceability perspective, there has been a large body of research [4, 7, 12, 16–18], but most focuses on the traceability of requirements, architecture and quality attributes. Recent research has looked at extracting traceability for feature models (a type of configuration model space) [8,15], but this has been achieved only through documentation, rather than by examining the multiple layers of the software preference space. We believe some of this work can be leveraged for configurability.

Finally, there has been a large body of work in the testing community that demonstrates the need for configuration-aware testing techniques [20–22, 28] and proposes methods to sample and prioritize the configuration space [5,10,26,28]. There has also been recent work that uses configurability as a way to avoid failures through self-adaptation [11]. But all of this work assumes that the configuration model is known (or is somehow extracted).

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a case study to evaluate the complexity that configurability adds for developers and testers. To do so we have studied three highly-configurable software systems. We have shown that our open source and industrial applications all have similar mechanisms for maintaining and modifying configuration options and presented an abstraction of this mechanism. We also see that there is no single (easily available) ground truth to determine the full possible configuration space. To this end we recommend merging multiple sources, developing cross-language analysis tools and providing traceability between the different configuration layers. We have also seen that the dynamic behavior can be difficult to understand, therefore we need to be cognizant of the lifecycle of the application to understand our exact configuration state during debugging. In future work we plan to implement some configuration merging techniques, and traceability links between the various layers. We also plan to examine a larger variety of highly-configurable systems to understand if the same model holds.

8. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation awards CCF-1161767, CNS-1205472, and the Air Force Office of Scientific Research award FA9550-10-1-0406.

9. REFERENCES

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *International Symposium on Foundations of Software Engineering*, FSE, pages 308–318, 2008.

[2] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Conference on Computer Supported Cooperative Work*, CSCW, pages 301–310, 2010.

[3] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *International Conference on Software Engineering, ICSE*, pages 261–270, Minneapolis, Minnesota, May 2007.

[4] J. Cleland-Huang, J. H. Hayes, and J. M. Domel. Model-based traceability. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 6–10, 2009.

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

[6] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. *SIGSOFT Software Engineering Notes*, 31(6):1–9, 2006.

[7] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *International Conference on Software Maintenance, ICSM*, pages 22–28, Sep 2013.

[8] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *The Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 290–300, 2013.

[9] LibreOffice. `http://libreoffice.org/`, 2013.

[10] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 243–253, 2011.

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

[12] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grunbacher, and G. Antoniol. The quest for ubiquity: A roadmap for software and systems traceability research. *International Requirements Engineering Conference, RE*, 0:71–80, 2012.

[13] S. Herbold, J. Grabowski, S. Waack, and U. Bünting. Improved bug reporting and reproduction through non-intrusive GUI usage monitoring and automated replaying. In *International Conference on Software Testing, Verification and Validation Workshops*, ICSTW, pages 232–241, 2011.

[14] W. Jin and A. Orso. BugRedux: reproducing field failures for in-house debugging. In *International Conference on Software Engineering*, ICSE, pages 474–484, 2012.

[15] L. C. Lamb, W. Jirapanthong, and A. Zisman. Formalizing traceability relations for product lines. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 42–45, 2011.

[16] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), Sept. 2007.

[17] J. I. Maletic and M. L. Collard. TQL: A query language to support traceability. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 16–20, 2009.

[18] A. Marcus, X. Xie, and D. Poshyvanyk. When and how to visualize traceability links? In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 56–61, 2005.

[19] Firefox. `http://www.mozilla.org/en-US/firefox/`, 2013.

[20] X. Qu, M. Acharya, and B. Robinson. Configuration selection using code change impact analysis for regression testing. *International Conference on Software Maintenance, ICSM*, 0:129–138, 2012.

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

[22] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *International Conference on Software Maintenance, ICSM*, pages 255–264, Oct 2007.

[23] A. Rabkin and R. Katz. Static extraction of program configuration options. In *International Conference on Software Engineering*, ICSE, pages 131–140, 2011.

[24] A. Rabkin and R. H. Katz. Precomputing possible configuration error diagnoses. In *Automated Software Engineering*, pages 193–202, 2011.

[25] S. K. Sahoo, J. Criswell, and V. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *International Conference on Software Engineering*, ICSE, pages 485–494, 2010.

[26] C. Song, A. Porter, and J. S. Foster. iTree: efficiently discovering high-coverage configurations using interaction trees. In *The International Conference on Software Engineering*, ICSE, pages 903–913, 2012.

[27] The Document Foundation. `http://blog.documentfoundation.org/2011/09/28/`, 2011.

[28] 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.

[29] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Symposium on Operating Systems Principles*, SOSP, pages 159–172, 2011.

[30] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *International Conference on Software Engineering*, ICSE, pages 312–321, 2013.

[31] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *International Conference on Software Engineering*, ICSE, pages 1074–1083, 2012.