Testing of User-Configurable Software Systems Using Firewalls

Brian Robinson
ABB Corporate Research
Raleigh, NC, USA
brian.p.robinson@us.abb.com

Lee White
EECS Department
Case Western Reserve University
Cleveland, OH, USA
lwhite4939@aol.com

Abstract

User-configurable software systems present many challenges to software testers. These systems are created to address a large number of possible uses, each of which is based on a specific configuration. As configurations are made up of groups of configurable elements and settings, a huge number of possible combinations exist. Since it is infeasible to test all configurations before release, many latent defects remain in the software once deployed. An incremental testing process is presented to address this problem, including examples of how it can be used with various user-configurable systems in the field. The proposed solution is evaluated with a set of empirical studies conducted on two separate ABB software systems using real customer configurations and changes. The three case studies analyzed failures reported by many different customers around the world and show that this incremental testing process is effective at detecting latent defects exposed by customer configuration changes in user-configurable systems.

1. Introduction

User-configurable systems are software programs (or groups of programs) created as general-purpose solutions to address a broad market need that many individual customers may have, each of whom have a smaller set of specific needs. The software provides the ability to solve such varying and diverse issues by executing a specific configuration created by a customer to address their specific needs.

Configurations direct the execution of the software and are made up of groups of configurable, library-like components, called configurable elements. These elements are only executed when they exist in a running configuration. In addition, each of these elements can contain a number of settings whose values further refine the actions that the element performs. Configuring systems such as these usually involves connecting or grouping instances of these elements and assigning specific values to their settings. These groupings can be set up either graphically or programmatically, depending on the implementation of the software and the needs of the specific market.

Settings are defined as values that exist inside a configurable element and are visible and changeable by the user. Changes to settings can be made when the system is offline or online, depending on the system. Settings resemble and act as parameters or attributes to the configurable elements they reside in. Similar to parameters and attributes, these settings can influence the specific behavior of the configurable element, such as selecting the specific internal code path that is executed or the return value that an internal algorithm computes.

Configurable elements, on the other hand, are defined as individual, encapsulated parts of the system that can be added to, or removed from, the system's configuration. These elements are represented in the source code as groupings of code and are similar to a class. A configurable element exists in a configuration as an instance, each having its own settings and memory space, just as creating an instance of a class does. Similarly, if a configurable element has no instances configured its code will never be executed.

Examples of these systems include real-time control systems and Enterprise Resource Planning (ERP) systems. Control systems are used to monitor and control the operation of many critical systems, such as power generation and pharmaceutical manufacturing. Users of these kinds of systems purchase a base set of software containing many different functions and rules which, either independently or in cooperation with the vendor, can be used to configure the system to the customer's specific process needs. ERP systems allow the unification of multiple data sources into one system that performs one or more specific business processes. Similarly, these systems contain base libraries and functions that make up the specific configurations used by customers.

Testing systems of this kind presents significant challenges to practitioners in the field, due to the large number of combinations possible. Currently, it is infeasible to completely test configurable systems before release [12, 13]. Due to this, the software contains many defects after is deployed to the field. These are referred to as latent defects and are the cause of significant cost and rework over the lifecycle of these software systems.

In practice, industry testers verify the system using common configurations that are created with expert knowledge by engineers that configure systems for customers directly. These configurations are created to test areas perceived to be high risk [2]. Once the system is verified using these methods, each new customer's configuration is very extensively tested. This testing is conducted when the software is first delivered, installed, and commissioned [3], and involves running the software thoroughly at the customer's site with both normal and error operating modes.

Once installed, users of these systems make many changes to their configurations during the software's lifecycle. These changes can cause failures related to latent defects that were not detected before the product was released. These latent defects are a major cause of concern for customers, as no code within the software changed, which customers often associate with the risk of new failures.

In this paper, a new approach to testing user-configurable software systems is presented, specifically aimed at finding latent defects that customers would detect. In this approach, each customer's initial configuration is thoroughly tested before the software is deployed. Additional testing of the software is postponed until customers make changes to their configurations, instead of trying to test as many other configurations as possible. By using a method completely based on user changes, only customer relevant defects will be detected and resolved. Data collected from failure reports at ABB show that configuration-based failures found by internal testing are only fixed 30% of the time, compared to non configuration-based failures which have an overall fix rate of 75% [24]. In addition, many of those configuration-based defects are postponed until customers in the field report them.

This new approach can be thought of a modified form of regression testing, and, as such, the main goal is to determine the testing required for each customer change to verify that the system still performs correctly. This method must also verify that no latent defects are exposed, as traditional regression testing methods assume no latent defects exist in the software.

The remainder of this paper is organized as follows. Previous work in configurable systems and regression testing is described in Section 2. Section 3 presents the new firewall model for configurations and settings. Section 4 describes how to create the new firewall. Section 5 presents the setup for the empirical studies and Section 6 presents their results. Finally, Section 7 discusses the conclusions of this research and future work needed in this area.

2. Background and Related Work

Many recent approaches to testing user-configurable software aim to run tests on configurations that span or cover the overall configuration space. One such technique combines statistical design of experiments, combinatorial design theory, and software engineering practices in an attempt to cover important, fault revealing areas of the software [11, 14, 16]. One study of open source software by NIST shows that these techniques can be effective when tests can cover a large number of configurable element pairs [19]. Another recent study shows a technique which prioritizes configurations, allowing earlier detection of defects but leading to a decrease in overall defect detection [15]. These studies were conducted on an open source software product and a small set of test cases from the Software-artifact Infrastructure Repository [20], respectively. While these approaches have shown positive results, the systems they were run were smaller and contained fewer configurable elements and settings than industrial software.

One common approach, both in industry [2] and in academia [13], aims to reduce the testing needed to all pairs of inputs together. This approach, called combinatorial interaction testing, has been shown to be effective for many systems. It is effective on systems where the majority of the defects are caused by two or fewer interactions. This approach does not scale well to systems with thousands of configurable elements and settings.

Another current approach relies on parallelism and continuous testing to reveal faults in a system. This system, named Skoll [22], runs multiple configurations in parallel on separate systems, allowing for a larger number of combinations to be tested. In addition, the system employs search techniques to explore the configuration space and uses feedback to modify the testing as it is being performed. Configuring many of these industrial user-configurable systems is a very time consuming task and generating enough configurations to allow this large scale parallel testing could be prohibitively expensive.

No previous research has looked at the impact that configuration changes have on exposing latent defects. One closely related area is regression testing. *Regression testing* involves selective retesting of a system to verify that modifications have not caused unintended effects and that the system still complies with its specified requirements [1].

Many *Regression test selection* (RTS) methods make use of control flow information to determine the impact of a change, such as [7, 9]. Besides control flow, many other dependency types are supported by RTS methods.

Using data flows allows the impact analysis to extend along data flow dependencies and identifies impacted areas that would otherwise be missed. These techniques, such as [10, 18], take longer to determine the impact of the change, but allow for increased defect detection when data flow dependencies are present. Additionally, research into scaling these techniques up to large systems exists [4, 23]. These RTS methods are all intended to detect regression defects coming from code changes within the software under test. When users change their configurations, these methods do not directly apply, as no changes were made to the software itself.

While unable to directly address configuration changes, the Firewall RTS method is used extensively within the proposed solution. While this is not the only RTS method that can be used, the firewall method was selected as it does not require existing dependency information. This method uses code-based change information to determine impact. The Traditional Firewall (TFW) uses control flow dependencies to identify impact, stopping one level away from the change [7]. An Extended Firewall (EFW) was created to identify impact when longer data flow dependencies are impacted by the change [18]. Firewalls also exist for dependencies dealing with global variables [6], COTS components [8], deadlock conditions [4], and GUI systems [5]. Due to space limitations, specific details on these firewalls are omitted from this paper. The steps needed to create each firewall can be found in [24].

3. The Configuration and Settings Firewall

In this section, the Configuration and Settings Firewall (CSF) is presented. This method addresses the problem of users changing system configurations and settings which may expose failures related to latent defects. Section 3.1 contains the overview of the approach. Changes to settings values are described in Section 3.2 and changes to configurable elements are presented in Section 3.3.

3.1 Overview

The CSF analyzes changes to the user's configuration, including both configurable elements and settings. This analysis is conducted whenever the users change the configuration or settings in the configuration that is running on the software. This is similar to current RTS methods, which are applied to software every time the code changes.

Latent software defects can exist in many different parts of the system. One common source of latent defects involve data flow paths, where a configurable element's specific action or output result is dependent on a value computed in a different configurable element. Another common source involves simple code paths, including paths inside a configurable element and paths involving system functions. Additionally, latent defects can remain hidden from view due to observability issues and can be exposed by changes in operator interactions, hardware, and other software running on the same shared resources can also expose latent defects. These additional change types are outside the scope of this research.

The CSF identifies the different types of changes that exist in the configuration, including both settings changes and configurable element changes. Each change will require code-based firewalls to be created to identify impacted dependencies. The specific firewalls needed for a given change depend on both the type of configuration change and the details of the code that implements the changed setting or element. The details on creating the CSF are presented in Sections 3.2 and 3.3.

Before the CSF is further described, two key assumptions must be made. The first assumption states that the focus of the CSF is on detecting latent software defects that exist inside the source code of the system that are revealed due to a change in the configuration. All other defects are outside the scope of this research. This includes detecting errors in the logic of the configuration itself. The second assumption is that the software and the configuration of the system should not be designed as a fully connected system, where every object or function has a dependency on every other object or function. If either case is true, that system will not benefit from the CSF or any other RTS method, since all changes will require a complete retest.

3.2 Settings Changes

A settings change is a change to a specific value that resides inside a configurable element that is both visible to and changeable by the user. Often, changes to these values can be made by either changing a configuration file or using a GUI interface. Since these changes can be made easily, customers often overlook the possible risks in these changes. In addition, some settings changes can be made to the system while it is executing, which can lead to serious failures if latent defects are exposed. Because of this risk, all settings changes should first be done in a test environment using the CSF to verify that new latent defects are not exposed.

Settings often affect the internal operation of the element they reside in. In addition, impact from settings changes can propagate to other external configurable elements through data dependencies. A common data dependency occurs when the output of one configurable element is affected by a settings change and is used as an input to another element connected to it. An example of a setting affecting the internal operation of a configurable element is shown in Figure 1

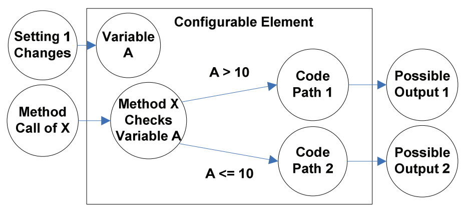

Figure 1. Example of a Settings Change

This figure shows an example of a settings change within a configuration. A change is made to Setting 1 in a configurable element. Setting 1 is represented in the code by Variable A, which gets assigned the value of Setting 1 when the system initializes. Variable A is used to determine if code path 1 or code path 2 is executed when method X is called. The value of Setting 1 was changed from ten to twenty. As a result of this change, code path 1 is now executed when method X is called, instead of code path 2. In this example, code path 1 has never been executed before in this configuration and could contain a latent defect. This setting change impacts both control flow and data flow dependencies, as it changes the path taken and the data output of the configurable element.

In real industrial systems, such as ERP and control systems, settings are usually represented as parameters, configuration files, and database values that are passed into or used by the various configurable elements in the system. Common ways that settings affect execution include defining boundary ranges for execution paths, assigning specific response actions for system events, and selecting between many options available to specialize the provided general solution.

3.3 Configurable Element Changes

A configurable element change involves adding or removing configurable elements from a preexisting configuration. These configurable elements provide functionality needed by users and are represented in the system as classes or functions, usually contained in code libraries. The code for a configurable element is never executed in the system unless instances of that configurable element exist in the running configuration, even though the code exists inside the system. Each configurable element can contain settings that control or impact its execution, although some configurable elements do not contain any.

There are three types of configurable element changes that can be made to a configuration, each of which can have a different impact on the system. The first type of change involves adding a new configurable element that does not exist elsewhere in the configuration. In this case, the code for this element has never been executed by this customer. This type of change has the highest potential risk for failures due to latent defects, as code that has never been executed in this customer's configuration is now activated.

The second type of change involves the addition of a configurable element that has been previously used in the customer's configuration, many times with different setting values than previously used instances. This is the most common configurable element change type found, as customers often extend an existing system by adding new instances of configurable elements they have used previously. Differences in the settings values between the new instance and those previously used are the main source of impact for this change type.

The final type of change involves removing a configurable element from an existing configuration. This is the least common type of change, as customers rarely remove previously running functionality. The most common reason for this type of change involves removing a configurable element and replacing it with a different element type. This type of change does not allow any new code to be executed and, in fact, removes code from within a previously executing configuration. Due to this, the highest risk for defects comes from the configurable elements that were using the outputs of the removed element.

In industrial systems, configurable elements are manipulated either graphically or programmatically. In the graphical case, configurable elements are represented by logical blocks and relationships between elements are represented as arcs connecting those blocks. For programmatically configured systems, configurable elements are often library functions or objects and relationships between them are invocations or instances of those functions or objects. Changes to these elements affect execution directly, by adding new code to execute, and indirectly, by changing data sent through the system used elsewhere by preexisting configurable elements.

An example configurable element change involves the addition of an element which shapes the input to smooth out any sudden value spikes due to a noisy input sensor. This added shaping algorithm is used elsewhere in the system configuration with different setting values. This type of change happens frequently, as customers change their configuration to correct imprecise physical behaviors that are not discovered until the actual plant is running.

4. Constructing the Firewalls

This section presents the details on how to create Configuration and Settings Firewalls. Section 4.1 presents the process for creating a CSF for a settings change and Section 4.2 presents the process for creating a CSF for each of the three types of configurable element changes.

4.1 Constructing a Firewall for Settings Changes

Constructing a CSF for settings changes involve following a set of steps, the totality of which define the firewall creation process. The general process for creating a CSF is shown in Figure 3. Each process step is represented by a circle, each valid transition is represented by an arrow, and any specific conditions that must be true to take a transition are listed as labels on the arrow. The two process steps inside of the box in the center of Figure 3 are general steps. These two steps are replaced with more detailed steps for settings changes and each of the three configuration change types.

Initially, customers have a previously created and tested configuration running in their environment. The customer then makes some changes to the configuration, involving one or more settings values, and saves it as a new configuration. If source code is available, the user can create the CSF for settings changes directly. If not, the user must send both the original and changed configuration to the software vendor for analysis and testing. This description assumes the vendor is doing the analysis, as many software vendors do not make source code available to the users of their systems.

The first step of the process involves determining the differences between the two configurations. The details for this comparison depend completely on the specific system being used. If the system is programmatically configured, simple text-based differencing tools are used to determine differences in the configuration code. For graphical or other types of configuration, custom tools are needed and are often supplied by the vendor. Only changes that affect the execution of the system are identified. Some changes, such as element names and comments, do not have any effect on the system and will not expose latent defects. Determining if the changes affect execution requires analyzing the source code of the configurable element to see how each changed setting is used. All changes that do affect execution are added to a list.

After the specific settings changes are identified, the source code representing each must be identified. This step is dependent on how configurable elements and settings are implemented within the source code. In a programmatic system, setting values are passed in to the system as parameters or configuration files, usually at system startup or in response to a defined event. In these types of systems, finding where settings values are used involves tracing parameter or input files from where they are accessed to their various usages in the code. If the system is graphically configured, a similar traceability is conducted starting at the GUI window and following the variable mappings into the source code to identify the usages of the changed settings values. Each area of code that uses the changed settings is marked as a code change.

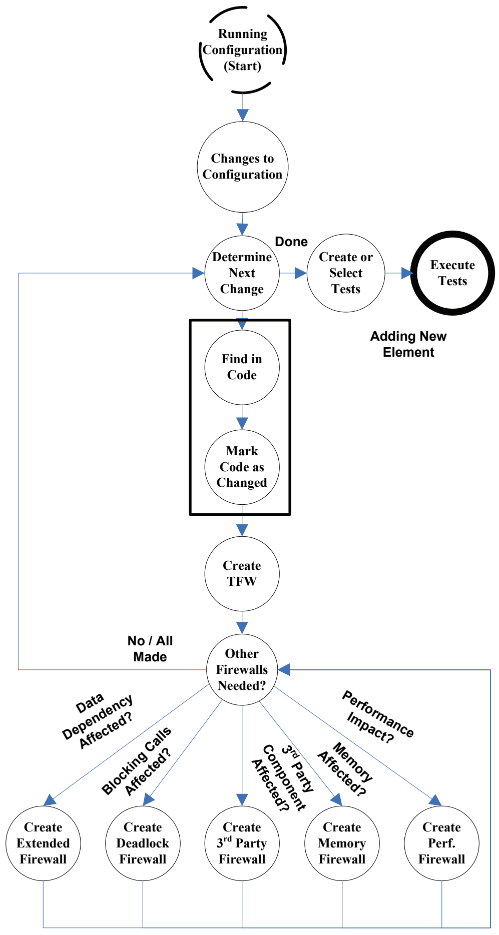

Figure 3. Process for Configuration Changes

Next, a Traditional Firewall is created for each section of code marked as changed. While creating the TFW, analysis is done to determine if the change impacts any other dependency types besides control flow. Common dependencies found in the systems studied include settings changes that affect data dependencies or impact the performance of the system. If any of these dependencies are found, the corresponding code-based firewalls are created. Some dependencies between configurable elements and with system functions are created dynamically when the configuration is loaded. These dependencies between configurable elements are only dynamic in the source code as they remain static throughout the entire execution of the system. Due to this, the configuration itself must be used when identifying these additional relationship types.

Each of the impacted areas identified by the various code-based firewalls has to be thoroughly tested to verify that no latent defects were exposed. These tests can be selected from previous testing done on these impacted areas or, if none exist, new tests must be created. Sources of reused tests include testing of previous changes for that customer, testing completed for other customers on the same areas, and tests that were used by the vendor for product release testing. Traditional, non-configuration based test techniques can be applied here, such as coverage and profiling techniques, equivalence classes, and boundary value analysis. Once the tests are ready, they are executed on the system and any failures logged.

4.2 Constructing a Firewall for Configurable Element Changes

Creating a CSF for changes in configurable elements follows the same process as for settings changes. Initially, a previously created and tested configuration is running in the customer's environment. The customer then decides that a change to the configuration is needed and adds or removes a configurable element from it, saving it as a new configuration. As with settings, this description will assume the vendor is doing the analysis and has access to the source code and both the new and changed customer configurations.

Next, the differences between the two configurations must be determined. This can be accomplished by using either a text based differencing tool for programmatically configured systems, or by using a custom tool provided from a software vendor for graphically configured systems.

After the differences have been identified, each change is categorized into one of the three possible changes types and its underlying source code is identified and marked as changed. Besides the configurable element itself, all relationships from the added configurable element to other parts of the system are marked as changed. These relationships include static relationships, such as accessing system functions, and dynamic relationships, such as other configurable elements using the output of the changed one.

Once all of the changes and relationships are marked as changed, a TFW is created for each. As with settings changes, analysis is done to determine if any of the changed code has dependencies that are affected by the change. Common dependencies in configurable element changes include impacting a data flow dependency, new code paths that can affect performance or memory, and interfaces with third-party components. Each dependency found has its respective code-based firewall created.

Each area identified by the firewalls as impacted must be tested. For new, added configurable elements, the testing focus is on covering the newly exposed code and the relationships between the new element and other areas in the system. When adding instances of previously used configurable elements, the testing focus is on the areas of code that deal with the differences in settings values between the new and previous usages. Finally, if configurable elements are removed, the testing focus is on affected external dependencies. After the tests are created, they are run and any failures found are logged.

5. Empirical Setup

In order to validate that the Configuration and Settings Firewall is effective and efficient, two case studies were performed on a GUI-based, real-time system configuration product. The system runs in the Windows OS on a standard PC. This product is implemented as a hybrid of OO-designed C++ code and procedurally designed C code. The system is made up of 5121 source files, 3229 classes, 39655 methods and functions, 767431 Executable Lines of Code (ELOC), 2398 configurable elements and 17 COTS components.

This software creates configurations for all of the products in the system. Configurations for this software are created graphically and compiled into binary. Customers then load these binary files into the various software products in the system.

Many customers are inherently secretive about their configurations. Currently, ABB has access to configuration data at two points in time. The first is the initial configuration created when the plant was first commissioned. The other configuration data comes from customers detecting field failures in the software. In this case, they share the configuration they used to expose the defect. While customers are inherently secretive about their configurations, they have expressed a willingness to share this data if it will lead to improvements in released software quality.

The first case study involves applying the CSF to a large number of past customer configuration changes that revealed latent defects. The customer-reported defects are then studied to see if they exist in the impact identified by the CSF. The goal of this case study is to determine the effectiveness of the change determination, code mapping, and impact analysis steps of the CSF at determining the correct areas of the software to test. This first study did not involve creating, selecting, or running any tests. This allowed the analysis to focus on the accuracy of the impact analysis independent of the quality of available and created tests.

The accuracy for the first case study is computed by analyzing the reported customer failures that are due to latent software defects and checking them against the impact identified by the CSF. If the defect exists within the impacted area, it is considered detected. If it exists outside the impacted area, the defect will be further studied to determine if it was related to the configuration change. If they are related, the defects are considered missed, and if not, they are considered outside of the scope of this firewall method and discounted.

The second case study takes a subset of the customer changes used in the first study and involves creating and executing tests for the impact identified by the CSF. The three goals of this study are to show the time required for creating the needed tests, to determine if the customer reported defect can be detected by testing, and see if any additional defects not initially reported by the customer can be detected.

The accuracy for the second case study is measured by the percent of customer-reported latent software defects that were detected by the testing identified by the CSF. Also, the time required to create and execute the tests is logged, representing the overhead associated in this process. Finally, any additional defects found by the testing are logged and compared to known defects in the system.

To prevent any bias in these two studies, no information about the customer-found defect was available at the time the CSF is created. Once the CSF is completed, this information is determined to evaluate the accuracy of the CSF method.

These case studies have two limitations. First, the configurations used by ABB customers are treated as trade secrets and there is no way to know exactly how all of the changes were performed over time. Since time sequence data for each change is not available, the total changes made to the customer's configuration are split arbitrarily into a set of smaller changes. This could lead to a larger amount of time for analysis and testing, due to creating overlapping firewalls between each set. The second limitation of the study is that the time to create and execute the tests is based upon a smaller number of changes. A larger study of test time will be created in future work.

6. Empirical Results

This section is split into Section 6.1, describing the results of the first case study, and Section 6.2, describing the results of the second case study.

6.1 First Case Study

This case study involves a number of different customer configurations and changes. A list of changes is created for each customer change. This list is then split randomly into smaller groups, each of which has a CSF created for it. The data collected for all customer changes are shown in Table 1. These data includes the number and type of created CSFs and code-based firewalls that were created for each customer change.

The first customer change studied for this GUI-based system involves a customer adding new graphical display elements into their configuration. The first set of changes contained three settings changes and the addition of two previously used configurable elements. The second set of changes contained two settings changes and the addition of one new configurable element. The corresponding CSFs were created and the details recorded in Table 1.

Next, the failures reported from the customer were analyzed. Each of the three added configurable elements exposed failures in the system. These failures were related to one latent defect in a support function used by these elements. This defect involved connecting configurable elements across different graphical pages of the configuration, by way of a reference that is implemented as a helper function available for all configurable element types. This helper function only produced a failure when called by the two types of configurable elements added by this customer. This defect existed in the impact identified by EFWs created by the CSF.

The second customer change studied for this GUI system involved a customer upgrading their Human Systems Interface (HSI) software. The customer changed the settings of existing configurable elements to take advantage of new features in their HSI. The first set of changes contained nine settings changes, each of which had settings change CSFs created for it. The second set contained the replacement of three configurable elements with ones containing additional functionality. These removals and additions, taken together, constituted an atomic change made in response to the HSI upgrade. The final set contains four settings changes. These settings changes affect the format of output data needed by the new HSI. The corresponding CSFs were created and the details recorded in Table 1.

Once complete, the reported failures were studied. There were four failures detected, each resulting in incorrect data being displayed on the new HSI. The failures caused values to be truncated to 14 characters, instead of the 16 characters stated in the requirements, and was caused by a single latent software defect contained inside the added configurable elements. This defect existed in the area of code identified as impacted by TFWs created by the CSF.

The third customer change studied involved adding five previously used configurable elements. These added elements represent redundant controller modules that were added for safety reasons. Once added, the customer exported the configuration to their HSI system. The operation failed to export all of the data to the HSI, as the export code contained a routine that counts the newly added redundant controllers incorrectly. This defect existed in the impact identified by EFWs created by the CSF.

Table 1. Summary of Case Study 1

	# of Settings Changes	# of Defects	# of Added Used CEs	# of Defects	# of Added New CEs	# of Defects	Analysis Time (Hours)	# TFWs	# EFWs	# Deadlock FWs	# 3rd Pty
HSI System											
Cust 1:	5	0	2	2	1	1	1.5	8	3	0	0
Cust 2:	13	0	0	0	3	1	2.5	16	4	0	0
Cust 3:	0	0	5	1	0	0	0.5	5	5	0	0
Cust 4:	0	0	10	1	0	0	3	18	0	0	1
Cust 5:	0	0	0	0	8	1	0.5	8	0	0	0
Cust 6:	25	1	0	0	0	0	4	25	0	1	0
Total:	43	1	17	4	12	3	12	80	12	1	1

The fourth customer change studied involves a change where the customer added ten previously used configurable elements to their configuration, eight of one type and two of another. These elements represent additional values needed by the operators and were added to the configuration loaded into the HSI. These added configurable elements are involved in a data dependency with a third party component database, which stores all of the data in the configuration. A failure was observed when a user-passed parameter is set to a negative value, as it is used as the index value to a database table. This defect existed in the impact identified by a COTS Firewall created by the CSF.

The fifth customer change studied involved a customer who added eight new configurable elements to the system. These elements represented an analog input module and data values connected to it. After these changes are made, the customer exports the configuration for use in another software product in the system. The export completes successfully, but when the configuration is loaded into the other product, eight failures are detected. The failures involved values from the newly added configurable elements being exported incorrectly. This defect existed in the impact identified by TFWs created by the CSF.

The final customer change studied included 25 settings changes. These settings changes affect the update rates of data being displayed. The change in timing was just enough to expose a latent deadlock to the customer when the configuration was changed. No other customers or testers had tested the system with those timing values before. This defect was identified by the Deadlock Firewall created by the CSF.

The final results of this study, detailed in Table 1, show that this method is effective at determining the impacted areas of the system that are at risk to expose latent defects due to a configuration change. The studied changes include 43 settings changes and the addition of 17 previously used configurable elements and 12 new configurable elements. The average time to create these CSFs manually was only two hours per change. These changes exposed 8 latent defects at customer sites, all of which were contained in the impact identified by the CSFs.

6.2 Second Case Study

The goal of this study is to run the required tests for a subset of the changes analyzed in the first case study. Few detailed tests exist in ABB to select for retesting configurable elements. As a result, the tests required for this study are created with exploratory testing [2], concepts from the Complete Interaction Sequences (CIS) method [5], and basic boundary value and code coverage metrics. The CIS method involves testing a required action by creating tests for all of the possible ways the GUI allows that action to occur.

As the tests were run on the system, certain measures were recorded. These are shown in Table 2. The first group of measures collects the time required to run these tests. One measure collected is a count of tests run. Another is analysis time, taken from the first case study, which represents the time needed to create the firewalls. The time required to run the tests, measured by a stopwatch, is logged as the third measure. After that, the total test time is calculated as the sum of the analysis and test times. Next, the original time is calculated by summing the time required to investigate and discuss this problem, including technical support, development, and management. By comparing the original time to the total time, a time savings is computed. This savings represents the time saved by using this method compared to the time required for the field-reported failures.

The second category of measures in Table 2 involves failures observed during the testing. First, the number of observed failures is counted. Each of the observed failures is split into two categories, known and new. New defects are not currently known by ABB and are not in the defect repository while known defects are.

The first change tested involved adding configurable elements which export values out of the GUI system and into the HSI. When the change was performed by the customer, values were truncated to fourteen characters instead of the required sixteen. When testing the impacted area for this change, four failures were observed.

Table 2. Results from Case Study 2

GUI System	# of Tests Run	Analysis Time	Test Time	Total Time	Original Time	% Time Saved:	# of Failures	# of Known Failures Found	# of New Failures Found	Reported Failure Found?
Change 1:	25	2.5	2.5	5	42	88.10%	4	3	1	Yes
Change 2:	18	1.5	2	3.5	51	93.14%	4	4	0	Yes
Total:	43	4	4.5	8.5	93	90.62%	8	7	1	100%

The first failure occurs when a settings value is being updated. If the user tries to switch GUI screens in the middle of updating the settings, they are prompted to save and, if the user selects cancel, all of the changes are lost. This failure was not found in the failures listed in the defect repository for this product, and is considered a new failure.

A second failure was found that occurs when the user enters 16 characters into a description field and tries to export the list of configurable elements. This export operation fails, as the exported list contains only 14 characters of the text. This failure matches the customer reported failure exactly, so it is counted as a known failure.

A third failure was detected when the customer configuration was first imported into the tool. The tool reported an error when this operation was first attempted, displaying only "Non-recoverable Error". This failure is a known error as it was detected internally by ABB when performing testing for a previous release.

The final observed failure occurred when a user exports the list of configurable elements. If the user selects an available option on the export dialog box, the resulting output contains no data regardless of what configurable elements are contained in the list. This failure was the same as one described in the defect repository originally reported by a separate customer one year after the release of this version. Therefore, it is counted as a known failure.

The second change tested in this study was the addition of three previously used configurable elements. These elements were connected across graphical pages by references. A failure was observed by the customer where the compiler did not complete correctly for the changed configuration. No error message was presented to the user and the only way to determine the compile failed is that no binary file was created.

When testing the impact of this change, four failures were observed. The first test involved simply compiling the project. This basic operation exposed a defect, where the compiler failed due to the customer adding three configurable elements. This first failure matches the original customer reported failure for this configuration change.

A second failure was observed when testing alternate ways to change settings in the added configurable elements. If a specific setting contains a value which comes from a configurable element on a different page of the configuration, then the entire program crashes when the configurable element is opened for change by the tool. This failure matches a failure found in the defect repository that was observed by a separate customer in the field.

An additional failure was observed while testing the export functionality. The system seems to export correctly, as a file is generated and no errors are detected. Once the file was opened, it was observed that the system failed to export all of the configurable elements and settings to the file and reported no errors. This failure was matched to a separate customer reported failure described in the first case study.

One final failure was observed when testing the impact of this change. Any textual changes to a setting value in the configurable element connected to the newly added element are not saved. This failure was found by another customer and existed in the defect repository at the time of this study.

Table 2 shows the final results of case study two. The average time required to create the CSFs and create and execute the required tests is 4.25 hours. Seven of these detected failures were reported by customers at a point in time later than the original configuration change. These represent defects that would have been found by ABB before future customers observed them. In addition, one new defect was found. These results show that testing the CSF-identified impact can detect the original customer-found failures as well as additional failures in areas around the change without a large effort required.

7. Conclusions and Future Work

User-configurable systems present many difficult challenges to software testers. Combinatorial problems prevent exhaustive testing before release, leaving many latent defects in the software after release. Customers are then at risk to exposing these defects whenever they make changes to their running configuration. The CSF was created as a solution to this problem that allows incremental testing of user-configurable systems based on configuration changes made in the field. Configuration changes are mapped to the underlying code of the configurable elements and settings. After this, tests are created or selected that cover the impacted areas.

Two case studies were performed on the CSF for this paper, aimed at showing its efficiency and effectiveness at detecting real customer found defects in deployed industrial systems. The results of the study show that each reported customer defect would have been detected if this method were used for that change. In addition, the analysis time required to create the CSFs is not substantial compared to the cost of diagnosing and fixing customer found problems.

The main area of future work is in initial release testing of user-configurable systems. Previous work in that area, such as [12, 13], shows some techniques which have been shown to work on systems with a small number of configurable elements and settings. These studies will need to be expanded for software with a larger number of configurable elements and settings, such as ERP systems and industrial control systems.

A further reduction in the testing needed for configuration changes can benefit from a better understanding of their execution in the field, using methods such as [21].

Automation is also needed for this method. Combining differencing tools with recent advances in semantic impact analysis techniques, such as [17], will allow many steps of CSF creation to be automated. The final goal would be a tool that enables customers to submit changes and get immediate feedback on the system impact.

8. References

[1] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," IEEE Standard 610.12, 1990.

[2] Kaner, C, Bach, J., and B. Pettichord. “Lessons Learned in Software Testing: A Context Driven Approach,” Wiley Publishing, New Jersey, 2001.

[3] Sommerville, Ian, “Software construction by configuration: Challenges for software engineering research”. ICSM 2005 Keynote presentation, Budapest, September 2005.

[4] L. White and B. Robinson, "Industrial Real-Time Regression Testing and Analysis Using Firewall," International Conference on Software Maintenance, Chicago, 2004, pp. 18-27.

[5] L. White, H. Almezen, and S. Sastry, "Firewall Regression Testing of GUI Sequences and Their Interactions," International Conference on Software Maintenance, 2003, pp. 398-409.

[6] H. Leung and L. White, "Insights into Testing and Regression Testing Global Variables," Journal of Software Maintenance, vol. 2, pp. 209-222, December 1991.

[7] L. White and H. Leung, "A Firewall Concept for both Control-Flow and Data Flow in Regression Integration Testing," International Conference on Software Maintenance, 1992, pp. 262-271.

[8] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "Applying Regression Test Selection for COTS-based Applications," International Conference on Software Engineering, May 2006, pp. 512-521.

[9] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," ACM Symposium on Principles of Programming Languages, January 1993, pp. 384-396.

[10] M. J. Harrold and M. L. Soffa, "Interprocedural Data Flow Testing," Testing, Analysis, and Verification Symposium, December 1989, pp. 158-167.

[11] Dunietz, I. S., Ehrlich, W. K., Szablak, B. D., Mallows, C. L., and Iannino, A. “Applying design of experiments to software testing.” International. Conference on Software Engineering, 1997, pp. 205–215.

[12] Cohen, M. B., Dwyer, M. B., and Shi, J. “Interaction testing of highly-configurable systems in the presence of constraints,” International Symposium on Software Testing and Analysis. July 2007, pp 129-139.

[13] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. “The AETG System: An Approach to Testing Based on Combinatorial Design,” IEEE Transactions on Software Engineering. July 1997.

[14] Kuhn, D., Wallace, D. and Gallo, A. “Software Fault Interactions and Implications for Software Testing.” IEEE Transactions on Software Engineering. June 2004, pp. 418-421.

[15] X. Qu, M.B. Cohen and K.M. Woolf, “Combinatorial interaction regression testing: a study of test case generation and prioritization,” International Conference on Software Maintenance, October 2007, pp. 255-264.

[16] Cohen, M. B., Snyder, J., and Rothermel, G. 2006. “Testing across configurations: implications for combinatorial testing,” SIGSOFT Software Engineering Notes November 2006, pp. 1-9.

[17] E. Hill, L. Pollock, and K. Vijay-Shanker. “Exploring the Neighborhood with Dora to Expedite Software Maintenance.” International Conference of Automated Software Engineering. November 2007.

[18] White, L., Jaber, K., and Robinson, B. “Utilization of Extended Firewall for Object-Oriented Regression Testing.” International Conference on Software Maintenance. September, 2005, pp. 695-698.

[19] Kuhn, D., and Reilly, M. “An investigation of the applicability of design of experiments to software testing.” NASA Goddard/IEEE Software Engineering Workshop. 2002, pp. 91–95.

[20] H. Do, S. G. Elbaum, and G. Rothermel. “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.” Empirical Software Engineering: An International Journal, 10(4):405–435, 2005.

[21] Dickinson, W., Leon, D., and Podgurski, A. “Finding Failures by Cluster Analysis of Execution Profiles.” International Conference on Software Engineering, May 2001.

[22] Adam Porter, Atif Memon, Cemal Yilmaz, Douglas C. Schmidt, Bala Natarajan, “Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance.” IEEE Transactions on Software Engineering, August 2007, 33(8), pp. 510--525.

[23] Orso, A., Shi, N., and M.J. Harrold, “Scaling Regression Testing to Large Software Systems.” Symposium on the Foundations of Software Engineering. November 2004, pp. 241-251.

[24] Robinson, Brian. “A Firewall Model for Testing User-Configurable Software Systems.” Ph.D. Dissertation, http://rave.ohiolink.edu/etdc/view?acc_num=case1206466905, Case Western Reserve University, 2008.