

# White-Box Approaches for Improved Testing and Analysis of Configurable Software Systems

Marius Nita      David Notkin  
Computer Science & Engineering  
University of Washington  
{marius,notkin}@cs.washington.edu

## Abstract

*There is a significant conceptual gap between the source code of a configurable system and the runtime behaviors of its individual configurations. In the source, configurations are woven together into a conceptually unified program. At runtime, however, they are largely treated as independent executables. This gap leads to static analyses that, by acting on the source representing the entire configurable system, yield imprecise results with respect to individual executables. Testing, in contrast, acts on individual executables without leveraging the configurable codebase per se. In this paper, we sketch a research path that seeks to narrow the configuration source-runtime gap, based on the observation that most configurations share significant amounts of source-level structure (hence “white-box”) with other, related, configurations. We seek to identify and exploit this structure to reduce analysis and testing effort by sharing analysis and test results among related configurations.*

## 1. Introduction

Roughly defined, a configurable software system is a shared code base from which a set of executable configurations can be produced. The system is characterized by a set of configuration options, each of which represents a dimension that can be configured. Each configuration option has a set of configuration values that can be selected. For example, a compiler may have “architecture” and “optimization level” as configuration options, with configurations values for each of these being {MIPS, x86,...} and {none, intra-procedural, inter-procedural, ...}, respectively.

There is a significant conceptual gap between the source code of a configurable system and the runtime behaviors of its individual configurations. The source code is written and maintained as one unified program in which configuration-specific code is identified using standard con-

trol constructs, such as `if` statements or `#ifdef` directives. At runtime, however, configurations are treated as stand-alone programs: the system can only be run in a particular configuration. Mirroring this dichotomy, static analyses, acting on the source, generally conflate all the configurations, as if they all may be active on any run, while testing, acting on the executables, considers only individual configurations.

Analyzing the full codebase almost always produces needlessly imprecise results with respect to specific configurations. While such results may have some uses, they are nonsensical for many analyses. For example, neglecting configuration information while computing a program slice will generally lead to a slice that includes code from superfluous and perhaps even conflicting configurations.

The testing problem is exacerbated by the fact that the number of individual configurations is usually exponential in the number of configuration options. Consequently, testing covers only a small part of a program’s configuration space. The standard approach is to invest resources predominantly in the assessment of a (characteristically small) set of established configurations determined to be of particular value. The (characteristically large) set of less-investigated configurations receives little attention. Problems with these configurations are often identified in the field by users rather than in the laboratory by the development team.

Our research focuses on approaches to increase confidence in the properties of software configurations while reducing the cost of doing so. We aim to narrow the gap between source code and runtime behavior by exploiting source-level structure (hence “white-box”) that is shared by sets of configurations. Our work is based on two central and related observations:

- Although configurations can in principle be arbitrarily far from one another in behavior, this is typically not the case — indeed, they are considered configurations because of their sharing of source code and of intended behavior. That is, the expectation is that the behavior

of two related configurations will usually be “close.”

- The source code that is shared across configurations — a resource that is rarely used for testing and analyzing more than individual configurations — affords an opportunity to extend evidence gathered during the test and analysis of established configurations to less-investigated configurations.

Our aim is not to exponentially reduce testing effort or to exponentially increase analysis precision. Rather, we aim to determine sets of *related configurations*, such that the testing or analysis results gathered from one configuration might apply, with some confidence, to its relatives at minimal or no cost to the programmer.

## 2. Background and Related Work

**Software Configurations** There is a relatively small set of results that focus on software configurations explicitly. Krone and Snelting applied *mathematical concept analysis* to infer the configuration structure of C programs [6]. When visualized, this structure offers insight into inter-configuration dependencies and the program’s configuration design space in general. Their implementation assumes that configurations are encoded using the C preprocessor. Configuration via command-line flags, preference files, environment variables, etc., is not supported.

Several research projects [13, 2, 12, 4] have focused on test coverage across configurations. They employ covering techniques — most prominently *covering arrays* — to automatically choose a manageable subset of the configuration space to test. This research does not address sharing results across related configurations and focuses solely on testing.

Our own prior work [9] addresses some of the software engineering issues associated with maintaining C code that can be configured to run on either big- or little-endian platforms. Using a programmer-provided specification and code written for one endianness, the code for the opposite endianness is automatically generated. The work does not address configurations beyond endianness, however. Neither does it address testing or analysis of configurations in any direct sense.

**Design for Configurability** There is a long history of design approaches intended to ease the development of configurable software systems. Dijkstra, in his discussion of THE, made an early suggestion about how hierarchical design may be consistent with “the aim that the software can be smoothly adapted to (perhaps drastic) configuration expansions” [3]. Parnas discussed this issue broadly, initially in his work on information hiding [11], then characterizing families of programs and advocating the use of layering to achieve configurability [10].

Object-oriented design has flavors of configurability as well: the core notion of inheritance explores ways of sharing code and behavior in a superclass while supporting variants through multiple subclasses. Open Implementation work [5] arose from a deep understanding of object-oriented approaches and was a major thrust in the definition of aspect-oriented programming, which isolates configuration-like alternatives in *aspects* that can be woven into the core program to provide related but distinct behaviors.

Software product lines are another design approach that allows particular kinds of configurability related to a focused market segment or mission. Testing and analysis of software product lines is not addressed aggressively in the literature; most focuses on processes for doing so [8, 1] rather than on testing or analysis per se.

These approaches, each in its own way, define mechanisms and design approaches to reduce the complexity of a software system intended to represent a set of configurations. They do not, however, directly address the issue of testing and analyzing the collection of configurations, either individually or as a whole.

**Configuration Management** Software configuration management is a term that has numerous definitions, most of which are consistent with “the practice of handling changes systematically so that a system can maintain its integrity over time. Another name for it is *change control*. It includes techniques for evaluating proposed changes, tracking changes, and keeping copies of the system as it existed at various points in time” [7]. The management of configurations, however, does not include specific approaches to testing and analyzing individual configurations or collections thereof.

## 3. A Sketch of the Configuration Space

Building and maintaining configurations is a largely *ad hoc* process. A diverse set of mechanisms are employed to store, read, and process configuration options and configuration code. Different systems read configuration options from command-line flags, configuration files, the environment, and dynamically from users via preference panes. Configuration-specific code can be written directly as part of the core source base or externally in the form of plugins.

A precise definition of software configurations is therefore elusive. However, we have identified two classes of configurations that describe the configurable systems we encountered in our studies and, we believe at least to some degree, configurable systems in general.

A *delta configuration*  $C_+$  augments a configuration  $C$  with an additional behavior, such that  $C_+$  does not exclude previous behaviors from  $C$ . For example, an optional compiler optimization can be encoded as a configuration (en-

abled by a command-line flag) that adds a new code transformation without removing existing ones.

Two configurations  $C_\alpha$  and  $C_\beta$  are *mutex configurations* if one *replaces* some behaviors in the other with new behaviors. Put another way, each of  $C_\alpha$  and  $C_\beta$  is formed by filling holes in a shared code skeleton. For example, a program could be configured to run under either of two different GUI APIs. The configurations are mutually-exclusive: exactly one must be active. Additionally, the mutually-exclusive behaviors added by two mutex configurations are often highly similar. For example, two mutex configurations targeting the two GUI APIs are highly similar in user-perceived behavior.

## 4. Formative Studies

In a recent graduate course, a group of students investigated test management in the GNU C Compiler and determined that the majority of configuration options were not covered by tests and that configuration options were added much more frequently than tests. They concluded, based on limited study, that there is a need for tools to manage the relationship between configurations and tests, to understand coverage, when tests are shared by configurations, etc.

Another group used static analysis to identify the data structures used to store configuration options and their values, to ultimately enable focusing on individual (or sets of) configurations at the source level. Their work identified a plethora of ways in which Unix programs are configured. Unsurprisingly, some of these mechanisms (such as command line flags) tend to allow straightforward static identification of configuration data structures, while others (such as environment variables) make the analysis difficult and perhaps intractable.

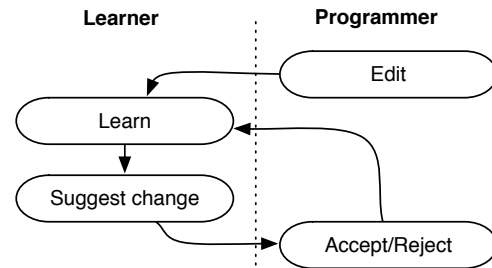
One study investigated the Google Gears browser plugin, focusing on two mutex configurations. One targets Internet Explorer (IE) and the other Firefox (FF). The study identified a *mapping* between the two configurations, associating types, data structures, and API calls on the IE side with corresponding elements on the FF side. It also identified exceptions: e.g., the IE counterpart of an FF-specific behavior would be incomplete or missing. The study suggests the need for tools to help identify and fix these exceptions.

## 5. Research Directions

Informed by our preliminary studies, we have identified a number of specific research problems that aim to exploit (a) sharing of code among configurations and (b) similarity of mutually-exclusive behaviors in mutex configurations to reduce testing, analysis, and maintenance effort in the development of configurable systems.

**Inferring and Using Configuration Mappings** As suggested in Sections 3 and 4, many mutually-exclusive (mutex) configurations are “equivalent” according to an *ad hoc*, programmer- and implicitly-defined notion of equivalence. Many development and maintenance tasks related to mutex configurations rely heavily on knowing the equivalence relation. Example tasks are verifying that the two configurations are equivalent, implementing a new configuration, or porting code and tests from one configuration to another. Because the equivalence is *ad hoc*, implicit, and likely only partially-known, these tasks are difficult and error-prone.

We are designing and building a system that encourages programmers to create a new configuration by copying, pasting, and changing an existing, relevant configuration. The development of the new configuration is a cooperative task between the programmer and a machine learner. The workflow is as follows:



The learner watches the programmer’s edits and maintains a *mapping* that contains partial knowledge of how the old configuration relates to the new one. For example, if the programmer replaces a line `List lst = new ArrayList();` with `int arr[] = new int[50];`, the system may hypothesize that `lst` and `arr` are related, that the elements stored in `lst` are related to those stored in `arr`, and that subsequent `List` operations on `lst` are likely to be replaced with equivalent array operations on `arr`.

The learner can use this knowledge to hypothesize and suggest further code changes. The programmer can optionally gain benefit by accepting or rejecting the suggestions, thus further training the learner and improving the quality of future suggestions.

The resulting mapping can be stored long-term and used for other purposes. For example, when a test is written for the old configuration, the mapping might contain enough knowledge to suggest how the test should be ported to the new configuration. For example, a test `f(); assert(lst.get(0) == 1);` might be ported as `g(); assert(arr[0] == 1);`, where `f()` and `g()` are stored in the mapping as related procedures. Similarly, when a future change is made to one configuration, the system can use the mapping to hypothesize and suggest (at least a sketch of) the equivalent change for the other configuration.

**Sharing Analysis Results Across Configurations** Current analysis and testing techniques consider either one con-

figuration at a time or all configurations at once. However, any given configuration is likely to share a large portion (usually the majority) of its source code with other “nearby” configurations. For example, a compiler configuration that enables an optimization shares the majority of its source code with the configuration that keeps the optimization disabled. We believe that we can translate this sharing of source code into reduced analysis effort.

Suppose configurations  $C$  and  $C'$  share a large amount of source code and we have run an analysis  $A$  on  $C$ . The crux of the problem is this: when running  $A$  on  $C'$ , we would like to re-analyze *as little as possible* of the source that was already subject to analysis when  $A$  was run on  $C$ .

Consider the following brief example:

1. $p = \&x;$	$\langle p \rightarrow x \rangle$	$\langle p \rightarrow x \rangle$
2. <code>if (C is on)</code>		
3. $p = \&y;$		$\langle p \rightarrow y \rangle$
4. $q = p;$	$\langle p, q \rightarrow x \rangle$	$\langle p, q \rightarrow x \rangle$

The code can be configured two ways, with  $C$  set to either `on` or `off`. The second column shows the per-statement results of a simple points-to analysis for  $C=\text{off}$ . The third column shows the results of the same analysis for  $C=\text{on}$ . We would like to obtain these latter results by analyzing *only* the statement on line 3, and quickly merging that result with the results of the analysis on  $C=\text{off}$ . Performing this merge is impossible without performing *at least some* re-analysis of the already analyzed code, however. For example, if line 4 were replaced by  $q=p; p=\&x;$ , the first analysis would yield the same results on line 4 but the second analysis would not, showing that analysis results alone are insufficient for a correct merge.

We are currently investigating *effect systems* that record enough information when performing the first analysis to enable a correct merge in the second analysis. We believe the effects to be as big as the shared code in a theoretical worst case but expect them to be much smaller in practice.

**Heuristics for Identifying Configurations** The basic building block for investigating configurations at the level of source code is the ability to separate particular (sets of) configurations from a configurable system. Particular problems include identifying the code associated with a set of options and their values, and identifying the options and values associated with a given line of code. We cannot expect to define an analysis that will handle all of the mechanisms used to program configurable systems (discussed in Section 3). However, our preliminary research suggests that the majority of configurable systems share a high-level structure that can inform a useful heuristics-based analysis.

A typical configurable system dedicates a subset of its code and data structures to processing and storing configuration options and their values. The configuration code is

executed at the beginning of a run to initialize the data structures, and may be (seldom) re-executed during the run as users reconfigure the program. Between (re)configurations, configuration data structures are read-only, and used to enable/disable configuration-specific code. Therefore, configuration data is long-lived: the ratio of writes to reads on configuration data structures is very small. We believe that, in general, this ratio tends to be a significant outlier when grouped with those of normal data structures, easing automated identification of configuration data structures and the code they enable or disable.

## 6. Conclusions

Based on a preliminary study, we have identified a number of directions for research targeted at reducing effort in testing, analyzing, and maintaining configurable systems. Our approaches are based on source code analysis and aim to take advantage of (a) the sharing of code among configurations and (b) the similarity of mutually-exclusive behaviors in many configurations.

## References

- [1] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *ISSTA'06 Workshop on the Role of Software Architecture in Analysis and Testing*, 2006.
- [2] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes*, 31(6):1–9, 2006.
- [3] E. W. Dijkstra. The structure of the “THE”-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.
- [4] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.
- [5] G. Kiczales. Beyond the black box: Open implementation. In *IEEE Software*, volume 13, pages 8,10–11, 1996.
- [6] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *ICSE '94*, 1994.
- [7] S. McConnell. *Code Complete*. Microsoft Press, 1993.
- [8] J. D. McGregor. Testing a software product line. Technical report, Carnegie Mellon, December 2001.
- [9] M. Nita and D. Grossman. Automatic transformation of bit-level C code to support multiple equivalent data layouts. In *International Conference on Compiler Construction*, 2008.
- [10] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Soft. Eng.*, SE-2:1–9, 1976.
- [11] D. L. Parnas. On the design and development of program families. *Software fundamentals: collected papers by David L. Parnas*, pages 193–213, 2001.
- [12] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA '08*, 2008.
- [13] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *SIGSOFT Softw. Eng. Notes*, 29(4):45–54, 2004.