Carnegie Mellon University

Ph.D. Thesis Proposal

Beyond Configurable Systems: Applying Variational Execution to Tackle Exponentially Large Search Spaces

Chu-Pan Wong
Carnegie Mellon University

December 3, 2019

Thesis Committee

Christian Kästner (Chair)
Carnegie Mellon University

Claire Le Goues
Carnegie Mellon University

Heather Miller
Carnegie Mellon University

Abhik Roychoudhury
National University of Singapore

Abstract

Variations are ubiquitous in software. Some variations are *intentionally* introduced, e.g., to provide extra functionalities or tweak certain program behaviors, while some variations are *speculatively* generated to achieve certain search goals, such as randomly mutating a buggy program to repair a bug. Although program variations provide great flexibility, their interactions are difficult to manage, as the number of possible interactions grows exponentially with the number of variations. Despite the challenges, there is increasing evidence showing that it is important to study interactions among variations in various domains, such as testing highly configurable systems, secure information flow tracking, higher-order mutation testing, and automatic program repair. In this thesis, we tackle exponentially large search spaces of interactions among variations.

Among existing approaches that study interactions among *intentional variations*, a recent dynamic analysis technique called variational execution has been shown to be promising. Variational execution can efficiently analyze many variations and keep track of their interactions accurately, by aggressively sharing redundancies of program executions. While existing use of variational execution has focused on *intentional variations*, we argue that variational execution is also useful for studying interactions among *speculative variations*, which remains an open challenge despite many years of research.

To study interactions among *speculative variations*, we set out to improve the scalability and extensibility of variational execution by using transparent bytecode transformation. With improved variational execution, not only can we extend existing work on intentional variations, but also open new avenues for analyzing speculative variations in higher-order mutation testing and automatic program repair.

Automatic program repair and higher-order mutation testing often use search-based techniques to find optimal or good enough solutions in huge search spaces of *speculative variations*. As search spaces continue to grow, finding solutions that require interactions of multiple variations can become challenging. To tackle the huge search spaces, we propose to encode the search problems as `Boolean` satisfiability problems, and use variational execution and SAT solving techniques to iterate all solutions efficiently. After identifying a complete set of solutions, we further study their characteristics to understand their nature and learn useful insights to inspire new ideas, such as more effective and lightweight metaheuristic search strategies.

Contents

Chapter 1

Introduction

Variations are ubiquitous in software, some are *intentionally* introduced and some are *speculatively* created. A typical example of *intentionally* introduced variations are program options, which are often controlled by command-line options or configuration files to trigger different functionalities or tweak existing features. Similarly, successful software frameworks tend to provide various extension points such as APIs for third-party developers to create extensions or plugins to enrich user experience. Beyond *intentional* variations, there are also variations that are created *speculatively* for various software engineering goals. For example, search-based automatic program repair techniques create hundreds or thousands of patch candidates while looking for potential patches [Monperrus, 2018]; and mutation testing approaches mutate an existing program to create different mutants to assess the quality of the existing test suite [Papadakis et al., 2019].

Although it is useful to create variations, whether they are *intentionally* introduced or *speculatively* created, their interactions are hard to manage. While *intentionally* introduced program options or framework plugins provide great flexibility, we risk the possibility that variations will create conflicts, especially those variations that are introduced independently by third parties, commonly known as the feature interaction problem [Calder et al., 2003; Nhlabatsi et al., 2008]. Conflicts among variations arise when one variation interferes with another in an unintended way, which is difficult to foresee even in small programs [Melo et al., 2016]. On the other hand, interactions among *speculatively* created variations are also of interests to researchers. A recent study by Zhong and Su [2015] shows that more than 70 % of bug fixes in practice require more than two repair actions (i.e., the interaction of more than two code changes), and Jia and Harman [2009] show that certain combinations (i.e., interactions) of first-order mutants are valuable for mutation testing, in that they denote more subtle bugs, reduce testing effort, and are less likely to be equivalent mutants.

There are several challenges posed by interactions among variations:

- The number of possible interactions is exponential to the number of variations, making systematic exploration of all possible interactions separately infeasible in most practical settings.
- Interactions of variations are difficult to track. The effects of interactions can easily propagate via control flow or data flow [Meinicke et al., 2016].

- Interactions are difficult—if not impossible—to foresee. For intentional variations, they are typically developed independently without any knowledge that other variations might exist. It is even more difficult for speculative variations as they are generated randomly.

Historically, researchers have done extensive research on tackling these challenges for *intentional* variations, as they are critical for software quality and information security in highly configurable systems [Nguyen et al., 2014; Austin and Flanagan, 2012]. In contrast, interactions among *speculative* variations are rarely studied, despite strong interests from researchers. For example, existing automatic program repair techniques can rarely generate successful multi-edit patches, bug fixes that require making multiple small changes to the buggy source code [Monperrus, 2018]. Mutation testing approaches typically create first-order mutants, mutated programs that have exactly one small change [Papadakis et al., 2019]. More broadly, the search-based software engineering community is concerned with finding a good balance of competing constraints by searching through different candidate solutions, but usually one at a time [Harman et al., 2012].

The goal of this thesis is to find effective ways to explore interactions among variations, transferring recent advances from *intentional* variations to *speculative* variations to inform existing research and inspire new applications in similar domains.

1.1 Analyzing Intentional Variations

Intentional variations typically manifest as program options or framework extensions, interactions among which could cause faulty behaviors or even security concerns. Researchers have proposed a broad spectrum of approaches to detect and manage their interactions. On the lightweight side, there are approaches that specifically target representative combinations of variations. For example, combinatorial testing covers n-way interactions among variations, by picking among all possible configurations a small set where each valid combination of n options appears at least once, where n is configurable and up to 6 in practice [Cabral et al., 2010; Cohen et al., 2007; Nie and Leung, 2011]. On the heavyweight side, there are verification approaches that use model checking or symbolic execution to statically analyze all possible interactions [Reisner et al., 2010; Sen et al., 2015; von Rhein et al., 2011]. In general, heavyweight static approaches can capture the entire space of possible combinations, but suffer from scalability issues due to state space explosion. In contrast, lightweight approaches cover the combination space in a less systematic way, but scale to realistic programs more easily in practice.

More recently, different researchers have independently proposed dynamic analysis techniques that seek to balance scalability and coverage of possible combinations [Nguyen et al., 2014; Meinicke et al., 2016; Wong et al., 2018; Austin and Flanagan, 2012]. Although called differently in different work, the idea is similar: Central to the scalability problem of most analysis techniques is the sheer quantity of possible inputs to the program, which include variations when performing analyses. By separating variations from other inputs, we can analyze interesting interactions among variations. Comparing to combinatorial testing, this line of work can explore interactions of any degree in a systematic and often efficient way by sharing similar executions. Based on this idea, researchers have proposed dynamic analysis techniques that analyze the effects of multiple variations by efficiently tracking variations at runtime. Researchers have applied these techniques to various scenarios, such as testing highly configurable systems [Nguyen et al., 2014; Kästner et al., 2012], understanding feature interactions and configuration faults [Meinicke et al., 2016, 2018], and monitoring information flow of sensitive data [Austin and Flanagan, 2012]. We call these techniques *variational execution* in this work, as they typically capture effects of variations at runtime.

1.2 Analyzing Speculative Variations

Speculative variations are changes made to an existing program automatically by other tools. For example, in automatic program repair and mutation testing, patch candidates and mutants are variations created speculatively for fixing bugs and introducing bugs, respectively. More broadly, the search-based software engineering community is interested in problems in which solutions are sought in a search space of candidate solutions, which are often variations of programs created speculatively [Harman et al., 2012].

Existing research on automatic variations often formulates the problem (e.g., fixing bugs or introducing bugs) as a search problem, in which optimal or near-optimal solutions are sought in a (often huge) search space of variations, guided by some metaheuristic search strategies and a carefully crafted fitness function that distinguishes good and bad solutions [Harman et al., 2012]. Although metaheuristic search strategies vary—such as genetic algorithms, hill climbing, and simulated annealing—existing approaches lean toward the lightweight side in the aforementioned solution spectrum, where the analysis can scale to realistic programs, but cannot explore the space systematically to uncover interesting interactions among variations. Existing heavyweight approaches for intentional variations transfer poorly to automatic variations, largely because the number of automatic variations is often much bigger than intentional variations due to the automatic and speculative nature, making heavyweight approaches such as model checking or symbolic execution even more difficult to scale. In this work, we propose to use variational execution to investigate interactions among (many) speculative variations, demonstrating automatic program repair and higher-order mutation testing as two important scenarios.

Recent successful applications in testing highly configurable systems and information flow tracking show that variational execution is promising in exploring large search spaces. However, speculative variations pose severe challenges to the scalability of variational execution techniques. On the one hand, the number of speculative variations tends to be much larger than intentional variations, mainly because they are generated speculatively. On the other hand, interactions among speculative variations can be complicated—in both control flow and data flow—because they are generated randomly without any consideration of modularity, which is usually considered best practice when introducing intentional variations manually [Parnas, 1972]. For example, we observed cases heavy interactions among hundreds of speculative variations cause a single local variable to have more than 15,000 possible values.

Since existing implementations of variational execution have issues in scalability and extensibility (more details in Chapter 3.1), we set out to implement a new variational engine that is more scalable and extensible to support our new applications, which becomes the foundation of this thesis. With more scalable and extensible variational execution, we use it to track fine-grained interactions among variations, while sharing commonalities to efficiently explore even exponentially large search spaces in many practical settings. At a high level, our approach proceeds in three steps:

- We encode variations (e.g., patch candidates or first-order mutants) as program options into a single meta program, using $boolean$ options to control inclusion or exclusion of individual variations.
- We use variational execution to explore combinations of variations *systematically* and *completely*. A secondary goal of accelerating the exploration of search space is also favorable, depending on how much sharing we can exploit during variational execution.
- We use variational execution to collect data and insights about all interactions, which can then be used to inspire new search-based strategies.

To gauge the potential of this work, we carefully analyze key elements that lead to successful applications of variational execution and show that those elements manifest in search-based automatic program repair and higher order mutation testing. The analysis of potential gives us confidence that this direction is promising. We envision that similar applications are feasible for other search-based problems, as long as they exhibit the key elements of applying variational execution (more in Chapter 2.4). We hope that this work can provide a new perspective of improving automatic program repair, mutation testing, and other related areas.

1.3 Thesis

In this section, we summarize the overarching goal of the proposed thesis, highlight main contributions, and discuss potential impact.

Thesis Statement: Variational execution can facilitate a systematic exploration of how variations interact in real-world software systems. Drawing inspiration from analyzing *intentional variations*, variational execution can be used to uncover interesting interactions among *speculative variations*. Demonstrating higher-order mutation testing and automatic program repair as two important applications, we show that variational execution can capture interactions completely and efficiently, and more importantly, reveal useful insights and knowledge that can be used to inspire new improved search-based strategies.

To support the thesis statement, we make the following contributions in the thesis.

- We propose a novel way of implementing variational execution using transparent bytecode transformation. Comparing to existing work, our approach automatically transforms existing programs without massive manual changes and produces transformed programs that are portable to all standard JVMs. We provide an open-source implementation called VarexC for the research community to explore new ideas.

- We evaluate VarexC with pre-established benchmark programs and show that VarexC has better performance and less memory consumption than the state-of-the-art. With improved performance and better scalability, our approach facilitates existing research on intentional variations, and more importantly, open the gate to exploring interactions speculative variations systematically.

- Speculative variations generated randomly could lead to heavy interactions that challenge scalability of variational execution. Fortunately, as we will discuss later, low-degree interactions among speculative variations are often preferable over high-degree ones. To this end, we propose a bounded BDD representation to restrict variational execution to explore only low-degree interactions.

- Drawing inspiration from *intentional variations*, we analyze two important applications of variational executions—testing highly configurable systems and information flow tracking—to identify ingredients of promising applications. These ingredients serve as a guideline for future research on applying variational execution.

- We use variational execution to find strongly subsuming higher-order mutants (SSHOM), a special kind of higher-order mutants that mitigate several open challenges of mutation testing research. Using previously used benchmark programs, we show that variational execution can identify, for the first time, a complete set of SSHOMs, greatly outperforming the state-of-the-art metaheuristic search in terms of time spent and SSHOMs found.

- By observing the identified complete set of SSHOMs, we identify a few patterns of how SSHOMs are commonly composed from first-order mutants. Based on these patterns, we further design a priority search and evaluate it on a different set of much larger benchmark programs. The results show that the patterns are effective in directing the search, again finding much more SSHOMs than the state-of-the-art metaheuristic search.

- Following a similar recipe, we apply variational execution to generate-and-validate automatic program repair. Despite active research, generating multi-edit patches remains one of the main challenges for automatic program repair. With variational execution, we have the unique opportunity to investigate how single-edit changes can interact to fix complex bugs.

- Reflecting on applying variational execution to higher-order mutation testing and automatic program repair, we perform a feasibility study to analyze how variational execution can be useful for opening new avenues for other similar areas, many of which fall under the umbrella of search-based software engineering.

With these contributions, we hope that the work in this thesis can improve software quality in general. We hope that our contributions to a more scalable variational execution technique can push the effort of quality assurance further toward more practical programs, for example, by scaling existing testing effort of highly-configurable systems and information flow tracking of sensitive programs. We hope that our contributions to analyzing speculative changes can reveal useful insights for research in mutation testing, automatic program repair, or more broadly search-based software engineering.

The remainder of the proposal is structured as follows.

- Chapter 2 introduces the ideas of variational execution and summarizes how it has been used in previous research to analyze interactions among *intentional variations*. Taking inspiration from two important successful applications—testing highly configurable systems and information flow tracking—we carefully analyze key ingredients that lead to promising applications of variational execution.

- Chapter 3 details our existing work on scaling up variational execution. By making variational execution more scalable and accessible, not only do we improve upon existing research on *intentional variations*, but also open the gate to a more systematic exploration of *speculative variations*. We discuss our existing work on bytecode transformation and early ideas of bounding variational execution.

- Chapter 4 describes how variational execution can be used to improve the search for strongly subsuming higher order mutants, a valuable kind of higher-order mutants that is difficult to find due to exponentially large search spaces.

- Chapter 5 outlines our ongoing effort of applying variational execution to generate-and-validate automatic program repair.

- Chapter 6 concludes the thesis proposal with a research plan.

Chapter 2

Background on Variational Execution

This chapter introduces the essential ideas of variational execution, specifically why it is effective in exploring interactions of variations. By analyzing existing applications of variational execution, we distill key ingredients for future applications.

2.1 Terminology

Variational execution has been explored in different domains in the past with different names and terminologies. To avoid confusion, we establish the terms that we will use heavily throughout the document.

Variation refers to an input that modifies the operation of a program. It is also called *option*, *feature*, or *flag* in the literature. We consider only `Boolean` variations in this work because they are common and easy to reason about with standard tools, but the techniques in this work can be extended to support other types of variations with *finite domains* (e.g., a predefined set of `Strings`, numeric values) by encoding them as `Boolean` options.

Intentional variation refers to an input that is introduced intentionally and mindfully, often by human developers. For example, program options are introduced purposefully to provide extra functionalities or tweak certain program behaviors.

Speculative variation refers to an input that is generated speculatively, randomly, and automatically by other tools. For example, mutation testing randomly creates small syntactic changes speculatively to introduce bugs in order to assess the strength of the existing test suite.

Configuration refers to a complete setting of all variations [Apel et al., 2013]. A configuration can be invalid if there are constraints among variations, such as having an option `A` to depend on another option `B`. Constraints like this are often specified manually based on domain knowledge.

Configuration space refers to all valid configurations, i.e., all possible settings of variations.

2.2 Variational Execution

Variational execution is a dynamic analysis technique that exploits sharing among similar *concrete executions*. Conceptually, variational execution abstracts over a finite number of *concrete executions* with minor differences that are caused by variations. There are two main concepts that distinguish variational execution from concrete execution: *conditional values* and *variability contexts*.

Conditional Value: The key idea of variational execution is to execute a program with *concrete values*, but support *multiple alternative concrete values* for different configurations. That is, whereas each variable has one concrete value in concrete execution (e.g., $x = 1$), the concrete value of a variable may depend on the configuration in variational execution—we say the variable has a *conditional value* [Erwig and Walkingshaw, 2013]. A conditional value does not store a separate value for each configuration in the configuration space (exponentially many), but partitions the configuration space into *partial spaces* which share the same value. That is, all configurations sharing the same concrete value are represented only once in the conditional value. Partial configuration spaces are expressed through propositional formulas over options, such as $(a \vee b) \wedge \neg c$ representing the potentially large set of all configurations in which configuration options a or b are selected but not c; a tautology (denoted as `true`) describes all configurations, a contradiction (denoted as `false`) none. Conditional values are typically expressed through possibly-nested choices over formulas (or if-then-else expressions), such as $x = \langle a, \langle \neg b \vee c, 1, 3\rangle, 2\rangle$, which means: x has the value 1 in the partial space $a \wedge (\neg b \vee c)$, 3 in $a \wedge \neg(\neg b \vee c)$, and 2 in $\neg a$. With this representation, we can reason about configuration spaces with SAT solvers and BDDs.

Variability Context: Variational execution uses conditional values with the notion of performing a computation conditionally in a *variability context*, similar to a path condition in symbolic execution: An operation will only modify values in the part of the configuration space indicated by the current variability context (that is, we conceptually split the execution). Again, formulas over configuration options are used to express the variability context.

Operations on *conditional values* under *variability contexts* can often be shared. If none of the used variables have alternative values, an instruction only needs to be executed once for all configurations (we say that we are executing under the `true` context). We begin execution in the `true` context, and only split into restricted variability contexts when configuration options influence execution—directly or indirectly. This conservative execution splitting strategy allows us to aggressively share executions that would otherwise be repeated once per configuration. This sharing avoids nonessential computations and makes variational execution efficient in many scenarios.

Example of Variational Execution

As an example, consider Listing 1 in Figure 2.1, a simplified implementation of a blogging system modeled after WordPress. [1] The blogging system has three variations, based on options for smiley rendering and inlining weather reports, which affect how HTML code is generated. In its current form, there is an issue: if both `SMILEY` and `WEATHER` are enabled, the replacement

[1]https://wordpress.org

Listing 1: Original version

```java
1  boolean SMILEY;
2  boolean WEATHER;
3  boolean FAHRENHEIT;
4
5  public String toHTML() {
6    String h = getHTMLHeader();
7    String c = getContent();
8    if (SMILEY)
9      c = c.replace(":]", "<img...>");
10   if (WEATHER) {
11     String w = getWeather();
12     c = c.replace("[:w:]", w);
13   }
14   String f = getHTMLFooter();
15   return h + c + f;
16 }
17
18 private String getWeather() {
19   float t = getCelsius();
20   if (FAHRENHEIT)
21     return (t * 1.8 + 32) + "°F";
22   else
23     return t + "°C";
24 }
```

Construct a conditional value containing two alternative values

Method `whenTrue` takes a `V` instance and returns the variability context under which the value is `true`

Construct a conditional value from two `V` instances

Method `smap` builds a new `V` by applying a function to elements selected by a context

Method `sflatMap` is similar to the `smap` except that the function should return a `V`

Construct a conditional value containing only one concrete value

Listing 2: Transformed version

```java
V<Boolean> SMILEY =
  new V<>(new PropExpr("SMILEY"), true, false);
V<Boolean> WEATHER =
  new V<>(new PropExpr("WEATHER"), true, false);
V<Boolean> FAHRENHEIT =
  new V<>(new PropExpr("FAHRENHEIT"), true, false);

public V<String> toHTML(PropExpr ctx) {
  PropExpr subCtx;
  V<String> h = getHTMLHeader(ctx);
  V<String> c = getHTMLContent(ctx);
  subCtx = whenTrue(SMILEY).and(ctx);
  if (subCtx.isSatisfiable())
    c = new V<>(subCtx,
      c.smap(subCtx, x->x.replace(":]","<img...>")),
      c);
  subCtx = whenTrue(WEATHER).and(ctx);
  if (subCtx.isSatisfiable()) {
    V<String> w = getWeather(subCtx);
    c = new V<>(subCtx,
      c.sflatMap(subCtx,
        x->w.smap(subCtx,
          y->x.replace("[:w:]", y))), c);
  }
  V<String> f = getHTMLFooter(ctx);
  return c.sflatMap(ctx,
    x->h.sflatMap(ctx,
      y->f.smap(ctx,
        z->y + x + z)));
}

public V<String> getWeather(PropExpr ctx) {
  PropExpr subCtx;
  V<Float> t = getCelsius(ctx);
  V<String> ret = new V<>(null);
  subCtx = whenTrue(FAHRENHEIT).and(ctx);
  if (subCtx.isSatisfiable())
    ret = new V<>(subCtx,
      t.smap(subCtx, x->x * 1.8 + 32 + "°F"),
      ret);
  subCtx = whenFalse(FAHRENHEIT).and(ctx);
  if (subCtx.isSatisfiable())
    ret = new V<>(subCtx,
      t.smap(subCtx, x->x + "°C"),
      ret);
  return ret;
}
```

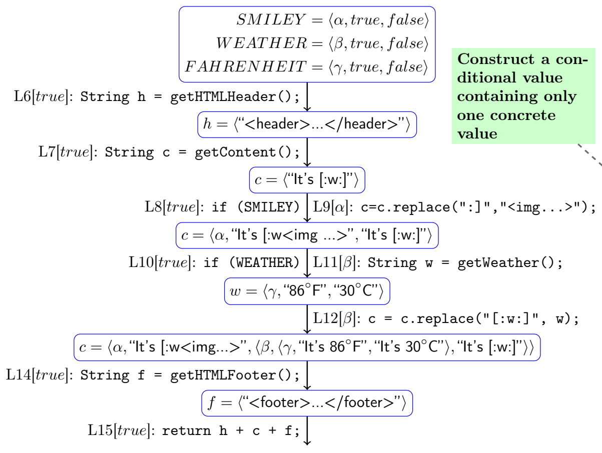

$SMILEY = \langle\alpha, true, false\rangle$
$WEATHER = \langle\beta, true, false\rangle$
$FAHRENHEIT = \langle\gamma, true, false\rangle$
L6[$true$]: String h = getHTMLHeader();
$h = \langle$"<header>...</header>"$\rangle$
L7[$true$]: String c = getContent();
$c = \langle$"It's [:w:]"$\rangle$
L8[$true$]: if (SMILEY)
L9[α]: c=c.replace(":]","<img...>");
$c = \langle\alpha,$ "It's [:w<img ...>", "It's [:w:]"$\rangle$
L10[$true$]: if (WEATHER)
L11[β]: String w = getWeather();
$w = \langle\gamma,$ "86°F", "30°C"$\rangle$
L12[β]: c = c.replace("[:w:]", w);
$c = \langle\alpha,$ "It's [:w<img...>", $\langle\beta, \langle\gamma,$ "It's 86°F", "It's 30°C"$\rangle$, "It's [:w:]"$\rangle\rangle$
L14[$true$]: String f = getHTMLFooter();
$f = \langle$"<footer>...</footer>"$\rangle$
L15[$true$]: return h + c + f;

Figure 2.1: Running example modeled after WordPress [Meinicke et al., 2016].
Listing 1 shows the original source code without variational execution.
Bottom left illustrates variational execution by showing the execution trace, where boxes represent relevant program states and arrows denote execution steps. The executed statements are displayed beside arrows, together with the variability contexts.
Listing 2 hints at our variational execution transformation, which will be discussed in detail in Chapter 3. The transformation is shown in Java for better readability.

of a smiley image takes precedence and breaks the expansion of weather information, resulting in outputs like "[:w😊".

For this example, let us assume that we have a specification of what a web page should look like. In order to ensure the absence of interaction bugs like this, typical testing techniques would try all configurations one by one, resulting into 8 executions of the same program in this case. Moreover, single executions alone reveal little information about the causes of interaction bugs, especially for cases where interactions are obscured by complicated control flow or data flow.

Variational execution is much more efficient for detecting and monitoring interactions. The execution trace in the bottom left of Figure 2.1 illustrates how variational execution explores all possible interactions among `SMILEY`, `WEATHER` and `FAHRENHEIT` in *a single run*. An execution trace like this can also be generated by logging and aligning concrete executions of all possible configurations, but Meinicke et al. [2018] showed that variational execution is much more efficient, sidestepping correctness and performance issues of alignment.

The execution trace in Figure 2.1 highlights why variational execution is efficient. After marking the three `boolean` fields as variations (e.g. via Java annotation), variational execution initializes them with *conditional values*, representing both `true` and `false`. The symbols α, β, γ denote the three variations respectively. Variational execution runs Line 6 and Line 7 *once* under the *variability context* of `true`, meaning that they are shared across all configurations. Sharing like this enables variational execution to explore large configuration spaces efficiently. To highlight sharing, we put all shared statements to the left of the arrows in the execution trace. The execution is split when it comes to the first `if` statement, where `c` is modified only under the *variability context* of `SMILEY`. At this point, the content of `c` changes from *containing one value for all configurations* to *having two alternative values depending on the variation* `SMILEY`, and this change is reflected in the *conditional value* assigned to `c`. Finally, variational execution is able to share the execution of common code again at Line 14, after splitting executions in two `if` branches.

This example illustrates the benefits of variational execution. We can spot the problematic interaction of `SMILEY` and `WEATHER` by inspecting the conditional value of `c`, as shown in the execution trace. In fact, all possible interactions are recorded and detectable by inspecting *conditional values* during the variational execution. All information about how variations interact can be obtained after one single run of variational execution, in contrast to exponentially many with normal execution, and the difference would still not be obvious without aligning all traces of normal execution. The effectiveness of variational execution comes from using *variability context* to manage splitting and sharing of executions.

Comparing to Symbolic Execution and Multi-Execution

Despite some similar concepts, there are important differences between variational execution and symbolic execution. A *conditional value* in variational execution is fundamentally different from a *symbolic value* in symbolic execution, in that the former represents *a finite number of concrete values* while the latter often represents *an infinite set of possible values* of a given data type. Unlike symbolic execution where operations are carried out on symbolic values, variational execution always computes with concrete values; symbols are used only to describe configuration spaces for distinguishing alternatives and for describing contexts, but never intermix with concrete values. For this reason, loop bounds are always known concrete values in variational execution, and we avoid other undecidability problems. By considering finite configuration spaces, reasoning about configuration space of conditional values involves *inexpensive and decidable* satisfiability checks with SAT solvers or BDDs, while symbolic execution is often limited by *expensive* constraint solving and the types of theories the underlying constraint solver supports. For instance, reasoning about array elements in variational execution is fast, because we know the concrete array indexes and elements, in contrast to symbolic execution where a symbolic array index can dramatically slow down constraint solving because it can potentially refer to every element in the array.

Furthermore, variational execution has different concepts of managing state and forking and joining when compared to symbolic execution. Symbolic execution often forks new states either completely or partially at every conditional branch, often resulting into exponentially many paths in practice, commonly known as the path explosion problem. For example, Meinicke et al. [2016] have demonstrated that state-of-the-art symbolic execution implementations for Java split off separate executions on variability and share only a common prefix. Some symbolic execution engines merge states from different paths to share executions after control flow decisions, for example, introducing new symbolic values or using *if-then-else* expressions to represent differences among values from different paths—different designs make different tradeoffs with regard to performance, precision, and implementation effort [Baldoni et al., 2018; Sen et al., 2015]. Variational execution uses a design that maximizes sharing. It maintains a single representation of program state throughout the execution where differences are represented at fine granularity (variables and fields) with conditional values. Program state is always modified under the current variability context, which is equivalent to merging states after every single statement.

Finaly, variational execution is fundamentally different from traditional approaches of multi-execution [De Groef et al., 2012; Devriese and Piessens, 2010; Hosek and Cadar, 2013; Kolbitsch et al., 2012; Su et al., 2007] and delta debugging [Kwon et al., 2016; Sumner and Zhang, 2013; Zeller, 2002] that execute programs repeatedly (either variants of the program or the same program with different inputs) to compare those executions to identify, for example, information-flow issues or causes of bugs. These kinds of approaches execute programs repeatedly in parallel and align those executions either afterward or through probes at specific points of the executions. In contrast, variational exploits sharing and allows to observe differences among executions during the execution.

2.3 Existing Applications

Variational execution has a number of existing and potential application scenarios in different lines of work. In each case, a program shall be executed for many variations, typically to observe the similarities and differences among configurations, often with the focus on *interactions* among variations.

In this section, we discuss two established use cases of variational execution—configuration testing and information flow tracking—followed by a brief summary of other closely related work.

2.3.1 Configuration Testing

Computer programs often come with variations that adjust functionalities on demand, in the form of command-line options, plugins, or features. Features offer great flexibility, but also incur risk of feature interaction problems [Calder et al., 2003; Nhlabatsi et al., 2008], where one feature interferes with another when used together. As discussed earlier, Figure 2.1 illustrates how variational execution is used for configuration testing. Comparing to brute-force testing of all configurations, which does not scale when the number of features is large, variational execution can efficiently execute the program once and record all possible interactions of features if there is sufficient sharing among executions.

To detect feature interactions, Nguyen et al. [Nguyen et al., 2014] applied variational execution to test WordPress with different combinations of 50 plugins, yielding 2^{50} different configurations. Their results show that variational execution can analyze the huge configuration space efficiently and exhaustively and identify a previously unknown feature interaction bug.

Along similar lines, Meinicke et al. [2016] and Kim et al. [2012] executed Java programs with configuration parameters (as used in our evaluation) to observe differences among different configurations. Given test cases to provide global or feature-specific specifications, variational execution can efficiently check such specifications by executing test cases over large configuration spaces [Nguyen et al., 2014; Kästner et al., 2012; Kim et al., 2012]. Soares et al. [2018] furthermore used differences among executions as clues to find suspicious feature interactions. Variational execution can further be used to explain the differences in program executions among multiple inputs [Meinicke et al., 2018], in line with delta debugging [Kwon et al., 2016; Sumner and Zhang, 2013; Zeller, 2002]. Reisner et al. [2010] used symbolic execution to also detect feature interactions, which however required a lot of effort (80 machine weeks to symbolically execute 319 tests with less than 30 configuration options for 10 KLOC programs) due to limited sharing abilities of symbolic execution Meinicke et al. [2016].

2.3.2 Information Flow Tracking

Information leaks in security-focused systems have gained substantial attention recently, especially leaks that are caused by subtle implicit information flow. In this line of work, a variation is different confidentiality levels over a sensitive input, which determine whether private values of that input or its effect can be observed. Dynamic information flow tracking struggles with implicit flows, especially from paths that are *not* executed [Austin and Flanagan, 2009; Chandra and Franz, 2007]. Austin and Flanagan [Austin and Flanagan, 2012] proposed a form of variational execution to track information flows precisely, called *faceted execution*, which separates the executions of high confidentiality input (denoted as H) and low confidentiality input (denoted as L) so that sensitive information in H is not affected by L.

The key idea is to compress information of both H and L into a *conditional value* (called faceted value in the original work [Austin and Flanagan, 2012]). When H and L have the same value, the executions are shared to reduce overhead. When H and L have different values, both values are accessed or updated according to some security-preserving semantics. If H and L have the same value later, the executions are shared again. Multiple principles (i.e., multiple pairs of H and L) are also supported and their interactions are explored at runtime. This line of work was later extended to support different languages and database systems [Austin et al., 2013; Schmitz et al., 2016; Yang et al., 2016; Schmitz et al., 2018].

Figure 2.2 shows an example of how to use variational execution to protect sensitive data. From a security perspective, the program input `x` should be hidden from public observers. Without variational execution, public observers can infer the value of `x` by checking the return value `z` because `x` and `z` should have the same value due to the implicit flow caused by the two `if` branches. The goal is to hide the secret value of `x` from public observers.

```
1 boolean f(boolean x) {
2   boolean y, z = true;
3   if (x)
4     y = false;
5   if (y)
6     z = false;
7   return z;
8 }
```

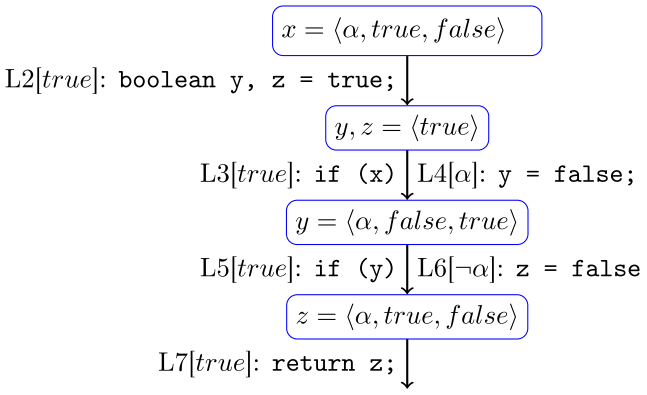

$x = \langle \alpha, true, false \rangle$
L2[$true$]: boolean y, z = true;
$y, z = \langle true \rangle$
L3[$true$]: if (x)
L4[α]: y = false;
$y = \langle \alpha, false, true \rangle$
L5[$true$]: if (y)
L6[$\neg\alpha$]: z = false
$z = \langle \alpha, true, false \rangle$
L7[$true$]: return z;

Figure 2.2: An example illustrating how variational execution can be used to handle implicit information flow.

With variational execution, x is initialized with a conditional value so that private observers see its real value $true$ and public observers see a fake value, using the symbol α to denote private and public observers. Using variational execution, values of H and L are separated safely. Finally, public and private observers see different values of z, so that the secret value of x is protected.

2.3.3 Other Applications

Researchers have explored ideas similar to variational execution in different lines of work to speed up computations. Variational execution can potentially be useful for these scenarios because of more aggressive sharing of similar computations. For example, Sumner et al. [2011] shares similarities among executions of simulation workloads and computes with several values in parallel. Wang et al. [2017] shares executions of mutated programs with equivalence modulo states in the same process and forks new processes only if there are differences in program states after executing mutated statements. Tucek et al. [2009] executes patched and unpatched programs together to share redundant computations when testing a patch. Since these lines of work do not look for interactions among variations, variational execution has the potential to scale such use cases to exploring interactions.

Similar ideas of sharing computations can be found also in approaches for model checking and symbolic execution [d'Amorim et al., 2007; Sen et al., 2015; von Rhein et al., 2011], specifically concepts to store variations as local as possible to increase sharing and facilitate joining. Such tools can potentially be used for similar purposes when differences among inputs are modeled as symbolic decisions, but all other inputs are concrete. However, as Meinicke et al. [2016] have shown, current approaches are less effective at sharing than the aggressive sharing in variational execution.

2.4 Key to Successful Applications

Despite the existing and potential applications of variational execution in different lines of work as discussed in the previous section, most research has focused on a single application, reinventing techniques independently. Moreover, researchers have mainly focused on analyzing *intentional variations*, changes or input differences that are made intentionally by human developers. We expect more application scenarios, such as automatic program repair and mutation testing, where this specific flavor of sharing computations with multiple concrete values is useful for exploring large configuration spaces. To predict applicability of variational execution in new domains, it is useful to elicit the key characteristics of the existing successful applications.

By analyzing how variational execution is applied to configuration testing and information flow tracking, we identified the following three main characteristics.

Finite Variations. The problem domain should have many but finite variations of interest to begin with. In configuration testing and information flow tracking, variations are different features and different private inputs, respectively. To justify the overhead of variational execution, the finite set of variations should depict a exponentially large configuration space that challenges existing lightweight approaches, such as metaheuristics search.

Interactions. Conditional values are especially useful for exploring interactions among variations at runtime. The overhead of variational execution is easier to justify when capturing all interactions among variations is invaluable. In configuration testing, developers are interested in the interaction of multiple options to detect bugs; in information flow tracking, interactions among multiple private inputs need to be tracked soundly to avoid leaking sensitive information in unexpected ways.

Sharing. Variational execution is effective if there is substantial sharing among executions of different variations and their interactions. Variational execution stores and computes all concrete values for all configurations, but it exploits shared values and shared operations to reduce overhead so that it can explore an exponentially large configuration space. If there is no sharing at all among executions, variational execution suffers from the same combinatorial explosion as a brute-force strategy. However, studies have shown that sharing is very common in practice for configuration testing [Meinicke et al., 2016; Reisner et al., 2010]. In information flow tracking, sharing is common in parts that are not affected by confidentiality levels or when outputs are independent of confidentiality levels. Interactions are common, but not among all variations at all times [Meinicke et al., 2016; Reisner et al., 2010]. That is, variational execution is effective in large search spaces when interactions among multiple variations are important but not all variations interact on all computations.

Research on speculative variations is emerging, but understanding interactions of speculative variations remain challenging. Using the three key criteria discussed above, we show that variational execution is useful for higher-order mutation testing (Chapter 4) and automatic program repair (Chapter 5). As discussed in the Introduction (Section 1.2), analyzing speculative variations requires variational execution to be more scalable, so we discuss our work of scaling variational execution in the next chapter before delving into the two new applications.

Chapter 3

Scaling Variational Execution

There exist different implementations of variational execution, but none of them is scalable or extensible enough. Thus, we set out to improve the scalability and extensibility of variational execution. With improved scalability, not only can we extend existing work on intentional variations, but also open new avenues for analyzing speculative variations, as we will show in Chapter 4 and Chapter 5.

The first part of this chapter covers our work on making variational execution faster with transparent bytecode transformation. We briefly discuss the motivation, approach, and results in this chapter. More details can be found in our OOPSLA'18 paper [Wong et al., 2018].

The second part sketches some early ideas of bounding variational execution, which is useful in cases where a full exploration of search space is infeasible but low-degree interactions are favorable.

3.1 Faster Variational Execution with Bytecode Transformation

Existing implementations of variational execution rely on either *manual modification to the source code* [Austin et al., 2013; Schmitz et al., 2016, 2018] or *modification to the language interpreter* [Nguyen et al., 2014; Meinicke et al., 2016].

On the one hand, variational execution can be implemented by writing the source code to use some libraries or programming language constructs, so that the programs compute with multiple values in parallel [Austin et al., 2013; Schmitz et al., 2018, 2016]. Implementations of this kind put a heavy burden on developers because the use of these libraries or language constructs usually obscures the original programs. Moreover, rewriting existing programs is often tedious and error-prone.

On the other hand, variational execution can be implemented by executing a normal program with a special execution engine, such as an interpreter that tracks multiple values in parallel with special operational semantics for each instruction [Meinicke et al., 2016; Nguyen et al., 2014; Austin and Flanagan, 2012]. Modified interpreters often suffer from a conflict between functionalities and engineering effort: It would be painstaking to modify a mature interpreter like OpenJDK, though it fully supports all functionalities of the language, whereas it takes less engineering effort to modify a research interpreter such as Java PathFinder [Havelund and Pressburger, 2000], which however provides incomplete language support and often mediocre performance. For example, VarexJ [Meinicke et al., 2016], the state-of-the-art for Java, is implemented on top of Java PathFinder's (JPF) interpreter for Java bytecode [Havelund and Pressburger, 2000]. For this reason, VarexJ inherits several limitations that restrict the programs it can analyze, such as incomplete language features (e.g., native methods), lack of advanced optimizations (e.g., just-in-time compilation), and slow performance due to meta-circular interpreting (i.e., JFP itself is yet another Java application).

We present a new way of implementing variational execution. Our approach sidesteps *manual modification to the source code* and *brittle modification to the language interpreter*. The key idea is to automatically transform programs in their *intermediate representation*. Specifically, we transparently modify Java bytecode automatically to mirror the effects of a manual rewrite. The resulting bytecode can then be executed on an unmodified commodity JVM.

Transforming programs at the intermediate language level has several benefits.

- Intermediate languages often have simple forms and strong specifications, both of which facilitate automatic transformation.
- Source code is not required, allowing us to transform also libraries used in the target programs. We can also analyze other programming languages that are compilable to the same intermediate language, such as Scala and Groovy for the JVM platform.
- Existing optimizations of the execution engine can be reused; in our case, our transformed bytecode can take advantage of just-in-time compilation and other optimizations provided by modern JVMs.
- Modifications at the intermediate level remain portable. Our transformed bytecode can be executed on any JVM that implements the JVM specification.

We implement a bytecode transformation tool that covers the entire instruction set of the Java language. [1] We prove that our automatic transformation is correct for all control-flow graphs and optimal with regard to sharing for a large subset. We further propose optimizations by performing data-flow analysis and using specialized data structures. An empirical evaluation with existing benchmark programs shows that our approach is up to 46 times faster while saving up to 75 percent of memory when compared to the state-of-the-art. In addition, our approach provides static guarantee of optimal sharing for 89.7 percent of the methods, and achieves optimal sharing at runtime for 99.8 percent of all other method executions.

For the purpose of this proposal, we discuss the motivation of using bytecode transformation in Section 3.1.1, provide an overview of the transformation in Section 3.1.2, and show some of the evaluation results in Section 3.1.3.

[1] A few minor exceptions are discussed in the original work [Wong et al., 2018]

3.1.1 Automated Rewriting

To motivate transforming bytecode automatically, we illustrate how the source code of our earlier WordPress example can be manually rewritten in Listing 2 of Figure 2.1. We show the rewrite in Java source code for better readability, as the same program in bytecode is typically longer and harder to read, obscuring the essential ideas of our rewriting. This manual rewrite in Listing 2 also highlights the key ideas (floating boxes in Figure 2.1) of our automated bytecode transformation.

We introduce *variability contexts* in all methods, represented by instances of the `PropExpr` class, which model propositional expression over configuration options. Variables are rewritten to use a new `V` type to store *conditional values*, either a single value for all configurations or different values for different configurations. To manipulate values in `V` objects, we use `smap` and `sflatMap` methods. The `smap` method applies a function to each alternative value of a `V`, and the `sflatMap` method does the same but allows to split configuration spaces, producing more alternatives. For example, the operation `v.smap(ctx, f)` on a conditional value `v` of type `V<T>` takes as arguments (1) a variability context `ctx` and (2) a function literal `f` of type `T => U`, representing the pending operation. It returns a new V instance of type `V<U>` that results from applying the function `f` to each concrete element that exists under `ctx` in `v` (recall that a conditional value stores concrete values along with the variability contexts under which they exist). The `sflatMap` method works similarly, but takes functions of type `T => V<U>`.

Note that the manual rewrite shown in Listing 2 is not exactly the same as our bytecode transformation, but close enough to show the key ideas. There are a few key points in this manual rewrite:

- Variables store conditional values, represented by `V` objects.
- Most operations on conditional values (e.g., calling the `replace` method, String concatenation) are redirected with `smap` and `sflatMap` and applied to all alternative concrete values. It fact, this replacement is sufficient for most bytecode instructions.
- Both the `if` branch and the `else` branch of an `if-else` statement are transformed into an `if` statement, a statement that checks whether there exists any partial configuration under which the surrounded code will be executed. If such a partial configuration exists, the surrounded code will be executed under a restricted variability context (e.g., Line 36–40).
- All method calls have one additional parameter `ctx`, representing the variability context under which this method is called. The variability context restricts all instructions of that method invocation. Also, multiple `return` statements in the same method are replaced with temporary assignment to a local variable, which is returned in the end of the method.

The transformation from normal code to variational code is nontrivial and obscures the program. For example, we almost double the size of Listing 1 in order to transform a simple example into a variational execution version. The introduction of `smap` calls and complicated control-transfer structures also obscure the intention of the original program, making it hard to understand and debug. This puts a heavy burden on the developers to understand variational execution and how to use it correctly. All of these issues can be resolved if we adapt an automatic transformation approach that is transparent to developers.

3.1.2 Overview of Bytecode Transformation

Our bytecode transformation approach has two orthogonal components—transformation of individual instructions and transformation of control flow. In a nutshell, we transform each bytecode instruction of the original program into *a sequence of bytecode instructions* to implement variational execution. The transformed instruction sequence has two main objectives:

First, the transformed sequence processes *conditional values* instead of *concrete values*. For example, the `fadd` instruction can compute the sum of two `float` values, but cannot sum up two conditional values. We transform the original `fadd` instruction to a new instruction sequence that can perform addition properly with conditional values, as shown in Figure 2.1 where the original floating point calculation in Line 21 is transformed to a `smap` call in Line 63.

Second, the transformed sequence enforces a set of transformation invariants. Ideally, the transformation of individual instructions should be local, meaning that the transformation of the current instruction should not be affected by other instructions around it. However, this locality assumption is not generally possible because an instruction often affects another instruction by leaving data on the *operand stack*. The operand stack is an internal JVM data structure that is used for exchanging data between instructions. To assist local transformation of individual instructions, we establish several transformation invariants, and strictly enforce them in the transformed bytecode sequence. These invariants help us establish a common ground about what to expect from the operand stack, enabling concise transformation of most instructions. In the original work [Wong et al., 2018], we discuss in great detail the transformation invariants and different ways of transforming different kinds of instructions.

While transformation of individual instructions enables the JVM to process *conditional values*, transformation of control flow makes sure the JVM can manage *variability contexts* for splitting and joining executions. For example, in a branching statement the condition may differ among configurations, such that we may need to execute both branches under corresponding variability contexts, but join afterward to maximally share subsequent executions.

We significantly change the way programs are executed to track and change variability contexts at runtime. As discussed earlier, variability contexts are propositional formulas over configuration options that describe the partial configuration space for which an instruction is executed, similar to path conditions in symbolic execution. Instructions executed in a variability context only have an effect on the state of that partial configuration space. The challenge is to propagate and change variability context to achieve a shared execution for all configurations with maximal sharing.

In the original work [Wong et al., 2018], we explain how we structure the program in blocks, and how we transfer control and contexts among these blocks. Subsequently, we also discuss and prove two important properties of our control flow transformation: (1) that variational execution preserves behavior of the original program and (2) that control transfer among blocks is efficient. For the purpose of this proposal, we omit technical details.

Table 3.1: Statistics about benchmark programs and performance comparison among JVM, VarexJ and VarexC. Statistics include lines of code, number of (`boolean`) options, and number of valid configurations. Numbers in bold denote the cases where VarexC or VarexJ outperforms brute force execution. The last three columns denote the relative speedup or slowdown.

Subject	LOC	#Opt	#Config	μJVM (in ms)	max JVM (in ms)	VarexJ (in ms)	VarexC (in ms)	VarexJ/ VarexC	VarexJ/ maxJVM	VarexC/ maxJVM
Jetty	145,421	7	128	949	1,246	**166,340**	**4,660**	36x	133x	4x
Checkstyle	14,950	141	$> 2^{135}$	811	946	*__89,366__	**3,825**	23x	94x	4x
Prevayler	8,975	8	256	13	44	**33,124**	**725**	46x	753x	16x
QuEval	3,109	23	940	0.03	0.38	2,354	1,244	2x	6,195x	3,274x
GPL	662	15	146	0.55	6.23	4,691	479	10x	753x	479x
Elevator	730	6	20	0.03	0.07	45	7.88	6x	643x	113x
E-Mail	644	9	40	0.02	0.06	21	6.19	3x	350x	103x

3.1.3 Evaluation Results

We evaluated our approach comprehensively in terms of execution time, memory usage, and sharing efficiency in the original work [Wong et al., 2018]. In this proposal, we highlight the performance results only. Specifically, we compare our implementation (named *VarexC*) against repeatedly executing the unmodified code in all configurations (brute-force execution) and against VarexJ [Meinicke et al., 2016], the state-of-the-art variational execution engine for Java, which executes bytecode with a modified JVM based on Java Pathfinder.

Table 3.1 summarizes the performance results. Comparing VarexC and VarexJ, we can see that VarexC outperforms VarexJ in all cases, with a speedup of 2 to 46. To investigate how useful configuration-complete analyses are in practice, we compare VarexC (and for comparison also VarexJ) with the time it takes to execute individual configurations, both average configurations (the μJVM column) and worst-case configurations (the max JVM column). The overhead of variational execution is generally high, which is explained both by the instrumentation overhead (creating and propagating conditional values, boxing, control-flow indirections, SAT solving at runtime), and by doing the additional work of executing all configurations. The overhead is usually only justified for large configuration spaces, and so VarexC (as VarexJ) outperforms the brute-force execution of all configurations only for Jetty, CheckStyle, and Prevayler.

We argue that the runtime overhead of VarexC is reasonable. Runtime overhead does increase for the cases where interactions of variations are heavily used, but the overhead amortizes quickly in large configuration spaces, which grow exponentially with the number of options, unless all options interact. More importantly, research shows that interactions do not increase with the worst-case exponential behavior in most cases [Meinicke et al., 2016; Reisner et al., 2010]. Even the academic programs that are designed to interact heavily are still well-behaved with plenty of sharing despite many interactions. Finally, we argue that the overhead is worthwhile if we consider the ability to identify all interactions among all options, for which the alternative is sampling only a small set of configurations.

3.2 Bounded Variational Execution

Although variational execution can aggressively share repetitive computations, it performs concrete execution underneath, and thus is subject to essential complexity of the program. For example, if a local variable `x` has thousands of alternative concrete values under different configurations, any operation on `x` still needs to be carried out thousands of times concretely. Recent research reveals that expensive interactions like this are rare among *intentional variations* because they are often created carefully with certain modularity [Meinicke et al., 2016; Reisner et al., 2010]. However, our early work on speculative variations indicates that expensive interactions can be common among *speculative variations*, mainly because they are generated randomly.

To avoid expensive interactions among *speculative variations*, we propose bounded variational execution, which limits the exploration of interactions up to a certain degree (see Chapter 3.2.1 for a formal definition). This way, we ignore the high-degree interactions that involve too many variations, but we can still *systematically* explore interactions that are within the bound by exploiting sharing in similar executions. We can also adjust the bound to incrementally scale up our analysis.

Bounded variational execution can be useful in cases where interactions involving many variations are less interesting. For example, automatic program repair techniques often consider hundreds of variations while trying to speculate a patch, but it is unlikely that we need all of them to patch a buggy program, in which case a complete rewrite of the program might be better for maintainability. In fact, state-of-the-art approaches rarely generate solutions that require any interactions among variations [Monperrus, 2018]. Interactions involving only a handful of variations (i.e., multi-edit patches) remain an open challenge for the automatic program repair community.

3.2.1 Preliminary Work

A naive way of bounding variational execution is to formulate the bound as a propositional formula that restrains variability contexts during execution, similar to using a feature model [Apel et al., 2013]. Our main concern with this approach is that the formula can get very big, to the extend that it might significantly slowdown operations on variability contexts, which happen very frequently during variational execution.

We perform bounding in binary decision diagram (BDD) to bound variational execution. A binary decision diagram is a compact way of representing a `Boolean` function [Bryant, 1986]. As discussed earlier, variational execution relies on heavy use of propositional formulas to represent variability contexts in order to keep track of changes made to partial configuration spaces. Our implementation VarexC uses BDDs internally to represent variability contexts because BDDs often have small memory footprint and support efficient operations (e.g., `AND`, `OR`) [Bryant, 1986].

To the best of our knowledge, there is no existing implementation of bounded BDD. In this proposal, we only sketch our preliminary work. We argue that a systematic research that generalizes bounded BDD is a valuable addition to the proposed thesis, but *optional* as we suspect that the underlying ideas are incremental and specific to variational execution. It is possible that similar ideas have been explored or published in other domains, such as hardware design where BDD is commonly used. We will conduct a systematic literature survey as the first step and acknowledge existing work (if there is any) in the final thesis.

BDD Basics

A binary decision diagram is a rooted, directed, acyclic graph that represents a `Boolean` function. A BDD consists of two kinds of nodes—decision nodes and terminal nodes. Decision nodes represent `Boolean` decisions, which are variations in the context of variational execution. Each decision node is labeled by a `Boolean` variable V_i and has two children nodes, often called the *low* child and the *high* child. An edge from V_i to its low (or high) child represents an assignment of `false` (respectively `true`) to V_i. Terminal nodes represent the evaluation results of a `Boolean` function, so there are only two types of terminal nodes, representing `true` and `false`, respectively.

Figure 3.1 shows a minimal BDD with only one decision. It represents the `Boolean` function $\neg\alpha$, as assigning `true` to α would lead to the `false` terminal node.

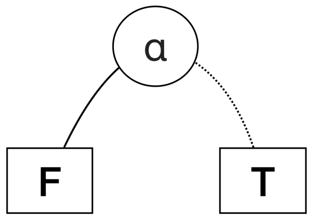


Figure 3.1: A simple BDD with one decision. The circle represents the decision α. The two boxes represent the terminal nodes `true` and `false`. A solid (or dashed) arrow represents an edge to high (respectively low) child.

In this work, we use the term BDD to refer to *reduced ordered binary decision diagram*, which is the most standard BDD in practice. The additional properties of being reduced and ordered enable efficient operations on BDD [Bryant, 1986].

Bounded BDD

We define the *interaction degree* of a `Boolean` function, denoted by $degree_{interaction}(f)$, as the *minimal* number of variables we have to enable in order to satisfy the function. Using a binary decision diagram to represent a `Boolean` function, the interaction degree is the minimal number of `true` edges (i.e., edges that point to a high child) among all paths that lead to the `true` terminal node. For example:

$$degree_{interaction}(x \land y) = 2 \tag{3.1}$$

$$degree_{interaction}((\neg x \land \neg y) \lor (\neg x \land y \land \neg z) \lor (x \land y \land z)) = 0 \tag{3.2}$$

In Equation 3.1, the interaction degree is 2 because setting both x and y to `true` is the only way to satisfy the `Boolean` function, while in Equation 3.2 we can satisfy the `Boolean` function by assigning `false` to all variables, hence enabling 0 variable.

Note that our definition of *interaction degree* is different from the standard definition of *degree* (denoted by $degree(f)$), which is the degree of the unique *multilinear polynomial* that represents f [Huang, 2019]. In the definition of *degree*, a `Boolean` function is first uniquely transformed into a multilinear polynomial form. The degree is then computed as the maximum number of distinct variables occurring in any monomial. For example, the *degree* the two `Boolean` functions above can be computed as:

$$degree(xy) = 2 \tag{3.3}$$

$$degree(1 - xy - xz - yz + 3xyz) = 3 \tag{3.4}$$

In Equation 3.3, there is only one monomial (i.e., xy) and it has two distinct variables (i.e., x and y). In Equation 3.4, the *degree* is determined by the monomial $3xyz$ as it has the maximum number of distinct variables.

For our purpose of bounding variational execution, our definition of *interaction degree* is more useful as it precisely captures the intuition of bounding a search space. When analyzing speculative variations, we often search for interesting combinations of variations. That is, we care about which variations should be enabled (i.e., assigned `true`), and care less about which of them should be disabled (i.e., assigned `false`). For this reason, our definition only concerns `true` edges that lead to the `true` terminal node.

As discussed earlier, our variational execution implementation uses BDDs internally to represent a partial configuration space. Since our goal is to explore all combinations of variations up to a certain bound (say n), variational execution moves forward with a variability context as long as the partial configuration space it represents includes any configuration that requires enabling fewer variables than n, hence the *minimal* number of variables we have to enable in our definition. On the contrary, if any solution to a variability context requires enabling more than n variables, subsequent variational execution will continue exploring in the partial configuration space that we deem expensive, and thus we can safely discard that variability context.

Existing algorithms for constructing a BDD start with a root node and then expand the graph in a top-down fashion until all paths to the terminal nodes are added. So the natural way of bounding a BDD is to count the number of true edges while constructing a BDD. When the number of tree edges reaches a predefined limit at a variable, we immediately make the `false` terminal node as its high child, but keep constructing the paths following its low child.

Research Plan

At the time of writing this proposal, we have a well tested implementation of bounded variational execution. We have plans for formalizing our bounding algorithm together with necessary correctness proofs. We also plan to explore other closely related topics, such as zero-compressed BDD and multi-terminal BDD. However, we argue that our current implementation is sufficient for the main purpose of the thesis, which is to explore interactions among speculative variations. A further optimized BDD solution and relevant publications are on our radar, as valuable addition to the proposed thesis, but not on the critical path.

Chapter 4

Higher-Order Mutation Testing

With a more scalable variational execution implementation (Chapter 3), we can systematically explore interactions among speculative variations. Specifically in the context of mutation testing, *speculative variations* are small syntactic changes to the program under analysis, and *interactions* are valuable combinations of these small changes that have been shown to denote more subtle errors.

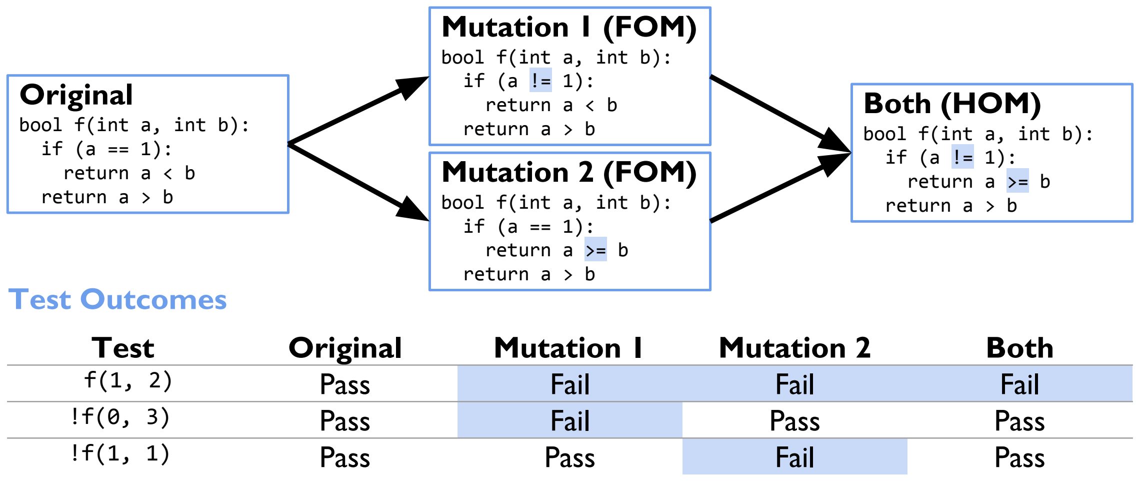
The work described in this chapter is excerpted from a conference submission under review at the time of writing. In this chapter, we provide an overview of our approach and highlight part of the interesting results.

4.1 Strongly Subsuming Higher Order Mutants

Mutation testing has been studied for decades for assessing and improving test suite quality [Papadakis et al., 2019]. In software engineering research, it is often used for various goals, such as evaluating a test suite, generating or minimizing a given test suite, or as a proxy for evaluating fault-localization techniques [Just et al., 2014b; Papadakis et al., 2019]. Mutation testing injects syntactic mutations into the program under test and runs the test suite to see if the test suite is sensitive enough to detect the mutations. Mutation testing has been shown to be promising, but also faces various challenges, such as the computational cost of repeatedly executing the test suite, effort required to identify equivalent mutants, and increased oracle cost when more test cases are required to increase the mutation score [Papadakis et al., 2019].

Higher-order mutation testing is the idea of combining multiple mutations with the goal of representing more subtle changes, more complex changes, or changes that better mirror human mistakes [Jia and Harman, 2009]. To that end, Jia and Harman [2009] distinguish *first-order mutants*, consisting of a single change, from *higher-order mutants* that combine multiple changes. Certain classes of higher-order mutants that produce non-trivial interactions of the changes are of interest. Among different classes of higher-order mutants, Jia and Harman [2009] highlight *strongly subsuming higher order mutants* because of their potential in denoting more subtle bugs. A subsequent study by Harman et al. [2014] further confirms various benefits of SSHOMs, as we will explain in Section 4.1.1.

Jia and Harman [2009] define a SSHOM as a higher-order mutant that can only be killed



Original

```
bool f(int a, int b):
  if (a == 1):
    return a < b
  return a > b
```

Mutation 1 (FOM)

```
bool f(int a, int b):
  if (a != 1):
    return a < b
  return a > b
```

Mutation 2 (FOM)

```
bool f(int a, int b):
  if (a == 1):
    return a >= b
  return a > b
```

Both (HOM)

```
bool f(int a, int b):
  if (a != 1):
    return a >= b
  return a > b
```

Test Outcomes

Test	Original	Mutation 1	Mutation 2	Both
`f(1, 2)`	Pass	Fail	Fail	Fail
`!f(0, 3)`	Pass	Fail	Pass	Pass
`!f(1, 1)`	Pass	Pass	Fail	Pass

Test	Orig.	Mut. 1	Mut. 2	Both	Failure Cond.
assert f(1, 2)	✓	✗	✗	✗	$m_1 \vee m_2$
assert !f(0, 3)	✓	✗	✓	✓	$m_1 \wedge \neg m_2$
assert !f(1, 1)	✓	✓	✗	✓	$\neg m_1 \wedge m_2$

Figure 4.1: Example with two mutations and corresponding test outcomes.

by a subset of test cases that kill all its constituent first order mutants. More formally, let h be a higher order mutant composed of first-order mutants $f_1, f_2, \ldots, f_n$, T_h the set of test cases that kill the higher-order mutant h, and T_i the set of test cases that kill the first-order mutant f_i, then h is a SSHOM if and only if:

$$T_h \neq \emptyset \quad \wedge \quad T_h \subseteq \bigcap_{i \in 1\ldots n} T_i \tag{4.1}$$

If we further restrict T_h to be a strict subset, we get a even stronger type of SSHOM, which we call *strict* strongly subsuming higher order mutant, denoted as *strict-SSHOM*. In other words, there must be at least one test case that kills a first-order mutant, but not the higher-order mutant. Thus, in a strict-SSHOM, multiple first-order mutants interact such that they mask each other at least for some test cases, making the strict-SSHOM harder to kill than all the constituent first-order mutants together.

In Figure 4.1, we show a concrete example of a (manually contructed) SSHOM, combining two first-order mutations. Intuitively, the first first-order mutant (replacing '==' by '!=') forces the execution to go into an unexpected branch, and the second (replacing '<' by '>=') inverts the return values. The two changes in control and data flow are easy to detect separately (i.e., killed by two test cases each), but the combination of them is more subtle and only detected by one test case. That is, this SSHOM is harder to kill than its constituent mutants.

4.1.1 Usefulness of SSHOMs

Recent years have witnessed increasing adoption of mutation testing in industry [Petrovic et al., 2018; Petrović and Ivanković, 2018], but there is also an increasing need for more selective and useful mutants to avoid wasting human effort and computation resources. For example, recent studies by Kurtz et al. reveal that most mutants are redundant, leading to inflated mutation scores and redundant test effort [Kurtz et al., 2015; Just et al., 2017; Kurtz et al., 2016].

The coupling effect hypothesis has been one of the main concerns limiting the exploration of higher-order mutation testing [Offutt, 1992]. However, there is increasing evidence showing that certain classes of higher-order mutants (e.g., SSHOMs) are more subtle and harder to kill, both theoretically [Gopinath et al., 2017] and empirically [Langdon et al., 2010; Jia and Harman, 2008, 2009; Harman et al., 2010; Omar et al., 2017]. Studies on real faults also indicate that a nontrivial 27 % of faults are not coupled to commonly used first-order mutants [Just et al., 2014b], and more than 70 % of real faults are caused by buggy code in more than 2 locations [Zhong and Su, 2015]. SSHOMs might thus be a promising avenue to better simulate real faults.

Researchers have also found that SSHOMs can mitigate some open problems of traditional mutation testing. First, a SSHOM is, by definition, at least as hard to kill as its constituent first-order mutants, which means a SSHOM is more subtle, representing deeper faults that require more careful testing [Jia and Harman, 2009; Harman et al., 2014; Jia, [n.d.]]. Second, a SSHOM can replace its constituent mutants without loss of test effectiveness (i.e., the test suite is just as sensitive), but with increased test efficiency (i.e., fewer mutants to examine). Suppose traditional mutation testing is used on the example in Figure 4.1, the two first-order mutants may inspire two separate test cases that kill them individually, while the higher-order mutant may inspire only one stronger test case. Reducing the number of test cases can drastically reduce test effort, as creating (strong) test cases requires nontrivial oracle cost from human developers [Jia and Harman, 2009]. Multiple studies observed significant test efficiency improvement achieved by replacing first-order mutants with second-order mutants [Polo et al., 2009; Mateo et al., 2013; Kintis et al., 2010; Papadakis and Malevris, 2010; Madeyski et al., 2014; Harman et al., 2014]. Harman et al. [2014] further report that using SSHOMs can simultaneously improve test efficiency by 14 % and test effectiveness by 5.6–12.0 %. Finally, multiple studies suggest that higher-order mutants are less likely to be equivalent mutants [Mateo et al., 2013; Kintis et al., 2010; Papadakis and Malevris, 2010; Madeyski et al., 2014].

In this work, we do not reevaluate the usefulness of SSHOMs for various applications (e.g., increasing test efficiency), which has been studied repeatedly and comprehensively in prior work. Instead, we focus on a technical problem: How to efficiently find SSHOMs.

4.2 Finding SSHOMs

A SSHOM is defined in terms of subset relation among mutants killed by a set of test cases. For a given set of first-order mutants, the search space is finite, though very large due to the combinatorial explosion. Since only few of the combinations are interesting and those are hard to find in vast search spaces, higher-order mutation testing has long been considered too expensive. Jia and Harman [2009] explored search techniques to find SSHOMs, finding that genetic search performs best. Although genetic search has been shown to successfully find SSHOMs, it requires considerable resources to evaluate many candidates, involves significant randomness, and cannot give guarantees of completeness, e.g., establish that no SSHOM exists or enumerate them all.

In this work, we develop a technique that can find a *complete* set of SSHOMs for *small to medium-sized* programs, which enables us to study characteristics of SSHOMs. We then use the characteristics to develop another new search technique that is lightweight, scalable, and practical. To this end, we proceed in three steps:

(1) Variational Search: First, for the purpose of studying SSHOM in a controlled setting, we develop a new search strategy `search`$_{var}$ that allows us to find a *complete* set of higher-order mutants for a given test suite and given set of first-order mutants in small to medium-sized programs. Specifically, we use *variational execution* (Chapter 2), a dynamic-analysis technique that explores many similar executions of a program. Conceptually, our approach searches all possible higher-order mutants *at the same time*, identifying, with a propositional formula for each test case, which mutants and combinations of mutants cause a test to fail. From these formulas, we then encode the search for SSHOMs as a *Boolean satisfiability problem* and use BDDs or SAT solvers to enumerate *all* SSHOMs. A complete exploration with variational execution is often feasible for *small to medium-sized* programs, because variational execution shares commonalities among repetitive executions and because modern SAT solving techniques are relatively fast. Though it does not scale, analyzing a complete set of SSHOMs for smaller programs allows us to study SSHOMs more systematically.

(2) Complete-Mutant-Set Analysis: Second, we study the characteristics of the identified higher-order mutants. Where previous approaches found only few samples of higher-order mutants, we have a unique opportunity to study the characteristics of higher-order mutants on a *complete* set. We analyze characteristics, such as, the typical number of mutants combined and their distance in the code. This helps us build a better understanding of higher-order mutants without potential sampling bias from a search heuristic. For example, we found that most SSHOMs are composed of fewer than 4 first-order mutants. Constituent first-order mutants tend to spread within the same method or the same class.

(3) Prioritized Heuristic Search: Finally, we develop a second new search strategy `search`$_{pri}$ that prioritizes likely promising combinations of first-order mutants based on the previously identified characteristics. The `search`$_{pri}$ is easy to implement and does not require the heavyweight variational analysis of `search`$_{var}$. Although it no longer provides completeness guarantees, it is highly efficient at finding higher-order mutants fast and scales much better to larger systems with tens of thousands of first-order mutants. We evaluate the new search strategy using a different set of larger systems to avoid potential overfitting. Our results indicate that the previously identified characteristics are useful in guiding the search.

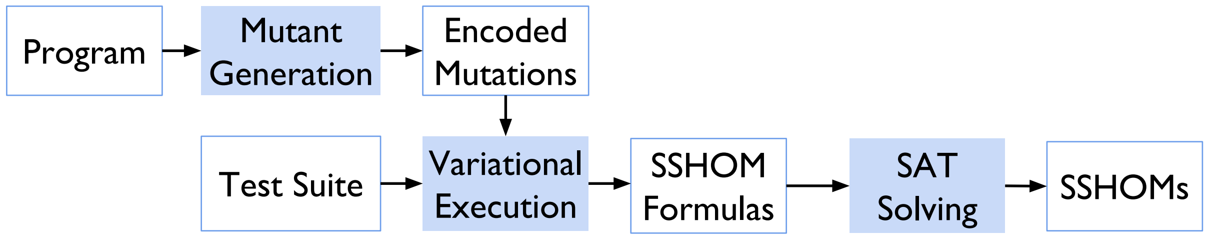


Figure 4.2: Overview of `search`$_{var}$. Shaded boxes represent steps.

```
bool f(int a, int b):
    if ( m1 ? a != 1 : a == 1 ):
        return  m2 ? a >= b : a < b
    return a > b
```

Listing 4.1: Mutated program with two first-order mutants encoded as runtime variability.

Our new search strategy can find a large number of SSHOMs despite an exponentially large search space, whereas existing search approaches can barely find any.

In the remainder of this chapter, we summarize the essential ideas of each step and highlight some of the interesting findings. More technical details can be found in our original work.

4.2.1 Step 1: Complete Search with Variational Execution (`search`$_{var}$)

In this step, we develop `search`$_{var}$ to compute a *complete* set of SSHOMs so that we can study the properties of SSHOMs. In Figure 4.2, we show the three main components, highlighted with shaded boxes.

First, given a program under analysis, we generate all first-order mutants upfront by applying our mutation operators exhaustively at every applicable location. We represent each mutant as a Boolean option and use a ternary conditional operator to encode the change. For example, in Listing 4.1, we show how we encode the two first-order mutants from Figure 4.1.

After encoding first-order mutants, we use variational execution as a black-box technique to explore which test cases fail under which combinations of first-order mutants. For a given test execution, variational execution will return a propositional formula representing exactly the combinations of options for which the test fails, which we illustrate for our running example in Figure 4.1 (last table column).

Finally, we collect all propositional failing conditions for all test cases and use them to search for SSHOMs by encoding the search as a Boolean satisfiability problem. Let T be the set of all tests, M be the set of all first order mutants, and f_t be the propositional formula over literals from M describing the mutant configurations in which test $t \in T$ fails (f_t is generated with variational execution, see above). As shorthand, let $\Gamma(m, t)$ be the result of evaluating f_t with first-order mutant m assigned to *true* and all other mutants assigned to *false*; in other words, whether or not test t fails for first-order mutant m. To identify SSHOMs, we encode three criteria:

1. The SSHOM must fail at least one test (i.e., must not be an equivalent mutant):

$$\bigvee_{t \in T} f_t \tag{4.2}$$

This check ensures that a mutant combination is killed by at least one test, encoding $T_h \neq \emptyset$ in Formula 4.1 (Sec. 4.1).

2. Every test that fails the SSHOM must fail each constituent first order mutant:

$$\bigwedge_{t \in T} (f_t \Rightarrow \bigwedge_{m \in M} (\neg m \vee \Gamma(m,t)) \tag{4.3}$$

If a given mutant combination (i.e., higher-order mutant) is killed by a test t, the same test must kill each constituent first-order mutant. That is, for all tests and first-order mutants, the first-order mutant must either be killed by the test ($\Gamma(m,t)$) or not be part of the higher-order mutant ($\neg m$). This is the encoding of $T_h \subseteq \bigcap_{i \in 1 \ldots n} T_i$ in Equation 4.1 (Sec. 4.1).

In addition, we can optimize for SSHOMs that are harder to kill than the constituent first order mutants, excluding those that are equally difficult to kill [Jia and Harman, 2009]. As discussed in Section 4.1, we call these *strict-SSHOM* and require a strict subset relation in Equation 4.1 (i.e., $T_h \subset \bigcap_{i \in 1 \ldots n} T_i$ rather than $T_h \subseteq \bigcap_{i \in 1 \ldots n} T_i$), which requires the additional encoded condition:

3. There exists a test that can kill all constituent first-order mutants but cannot kill the strict-SSHOM.

$$\bigvee_{t \in T} \big(\neg f_t \wedge \bigwedge_{m \in M} (\neg m \vee \Gamma(m,t))\big) \tag{4.4}$$

To find SSHOMs and strict-SSHOMs, we take the conjunction of Equations 4.2–4.3 and 4.2–4.4, respectively, and use BDD or SAT solver to iterate over all possible solutions. For example, if our approach returns a satisfiable assignment in which m_1 and m_3 are selected and all other mutants are deselected, then the combination of m_1 and m_3 is a valid (strict-)SSHOM.

We use BDD to get satisfiable solutions by default, as *VarexC* uses BDDs internally to represent propositional formulas. While constructing BDDs can be expensive during variational execution, getting a solution from a BDD is $\mathcal{O}(n)$, where n represents the number of Boolean variables [Bryant, 1986]. In some rare cases where we cannot compute a BDD due to issues like insufficient memory, we fall back to using a SAT solver. With a SAT solver, we ask for one possible solution, then add the negation of that solution as an additional constraint before asking for the next solution, repeating the process until all solutions are enumerated. Using this encoding, we can usually efficiently enumerate *all* possible SSHOMs for the given set of first-order mutants and the variational-execution result of a given test suite.

Table 4.1: Subject Systems and Found (strict-)SSHOMs; the last three subjects and the *priority* strategy are used only in Step 3 of this paper.

				Found SSHOM				Found strict-SSHOM			
Subject	LOC	Tests (%used)	FOMs (%used)	Var	Gen	BF	Pri	Var	Gen	BF	Pri
Validator	7,563	289 (83%)	1941 (98%)	$1.34 * 10^{10}$	4,041	273	36,995	281	0	4	10
Chess	4,754	847 (84%)	956 (29%)	3268†	484	19	16,403	216	0	6	24
Monopoly	4,173	99 (89%)	408 (90%)	818	81	349	817	43	4	15	43
Cli	1,585	149 (95%)	485 (88%)	376	309	326	369	21	18	21	21
Triangle	19	26 (100%)	128 (100%)	965	949	493	965	6	6	6	6
Ant	108,622	1354 (77%)	18,280 (92%)	-	1	0	44,496	-	0	0	61
Math	104,506	5177 (79%)	103,663 (100%)	-	0	0	390,533	-	0	0	2,830
JFreeChart	90,481	2169 (99%)	36,307 (99%)	-	0	6	576,725	-	0	0	513

LOC represents lines of code, excluding test code, measured with *sloccount*. Tests and FOMs report the total number of test cases and first-order mutants, respectively, with the actually used ones reported as percentages in parentheses. Var, Gen, BF, Pri denote our approach (Step 1, search$_{var}$), the genetic algorithm (search$_{gen}$), brute force (search$_{bf}$), and our prioritized search (Step 3, search$_{pri}$) respectively.
† incomplete results, solutions found with SAT solving within the 12 hours budget.

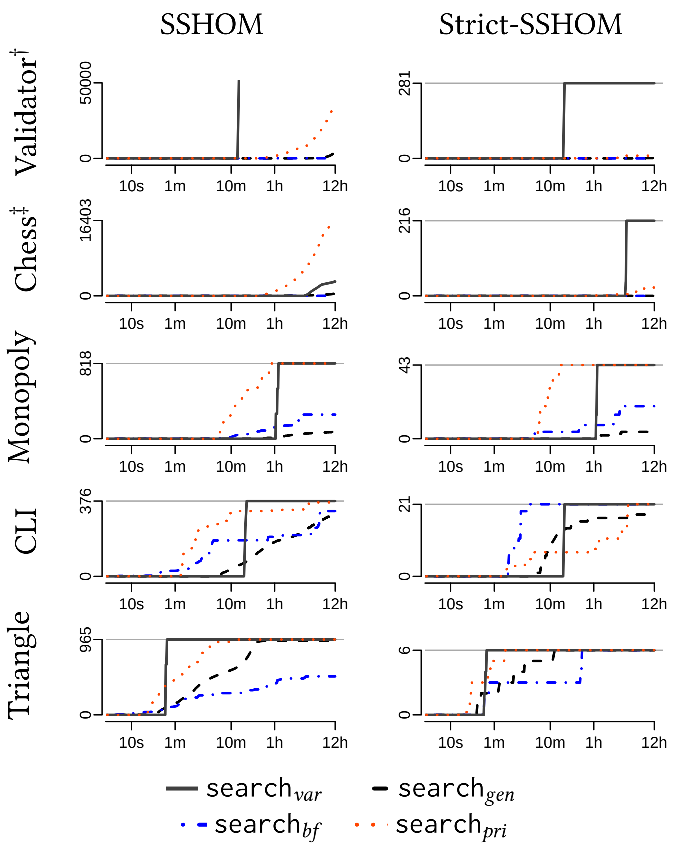
Evaluation

In addition to using search$_{var}$ to get a complete set of SSHOMs, we compare efficiency and effectiveness of search$_{var}$ against the existing state-of-the-art *genetic search* (search$_{gen}$) and a baseline *brute-force* strategy (search$_{bf}$), based on subject systems previously used in evaluating the genetic search strategy [Harman et al., 2014], as shown in the first half of Table 4.1.

Table 4.1 shows the number of (strict-)SSHOMs found with all three search strategies within the 12 hour time budget and Figure 4.3 plots the numbers of found (strict-)SSHOMs over time. Note that by construction, if search$_{var}$ terminates (all cases except *Chess*, where solving the satisfiability problem takes considerable time), it enumerates *all* SSHOMs, thus provides an upper bound for other search strategies—without search$_{var}$ this upper bound would not be known.

These results show clear trends: search$_{var}$ requires a relatively long time to find the first SSHOM, because variational execution must finish executing all tests for all combinations of first-order mutants. However, once variational execution finishes, it can enumerate *all* SSHOMs quickly by solving the Boolean satisfiability problem. Variational execution takes longer with more and longer test cases and with more first-order mutants, but still outperforms a brute-force execution by far, indicating significant sharing, as found in prior analyses of highly-configurable systems [Meinicke et al., 2016; Wong et al., 2018; Nguyen et al., 2014].

In contrast, search$_{gen}$ (i.e., genetic search) and search$_{bf}$ (i.e., brute-force search) can test many candidate SSHOMs before variational execution terminates and sometimes finds some actual SSHOMs early, but both approaches take a long time to find a substantial number of SSHOMs and miss at least some SSHOMs in all subject system within the 12h time budget given. In some systems with moderate numbers of first-order mutants, search$_{bf}$ is fairly effective as it systematically prioritizes pair-wise combinations which are more common among SSHOMs than combinations of more than two mutants, as we will discuss.



† We cap the plot for Validator since there are 13.4 billion SSHOMs; ‡ we could not enumerate all nonstrict-SSHOMs for Chess due to the difficulty of the SAT problem and report only those found within the time limit

Figure 4.3: SSHOM and strict-SSHOM found over time in each subject system, averaged over 3 executions. Note that time is plotted in log scale as most SSHOMs are found within the first hour.

4.2.2 Step 2: SSHOM Characteristics

In a second step, we study the characteristics of (strict-)SSHOMs, with the goal to inform subsequent heuristic search strategies (Step 3) and future research in general. Using the *complete* set derived for the subject systems in the previous step, rather than a (potentially biased) sample of SSHOMs, we can study characteristics with higher confidence.

We explored the dataset in an iterative exploratory fashion, focusing primarily on characteristics that may guide future search strategies, such as specific composition patterns and proximity of constituent first-order mutants for the set of all higher-order mutants. Kurtz et al. [2016] argue that mutation operators should be specialized for individual programs, so we focus on high-level characteristics that are largely independent of specific mutation operators to avoid overfitting.

Mutation Order. SSHOMs and strict-SSHOMs are typically composed of only few first-order mutants. Overall, over 90 % of all SSHOMs and strict-SSHOMs are composed of at most 4 first-order mutants, indicating that subtle interactions are mostly caused by few first-order mutants. Although few SSHOMs were composed of up to 6 first-order mutants (in Chess and Triangle), such cases are rare, especially for strict-SSHOMs.

Equivalent Test Failures. In multiple subject systems, many SSHOMs and strict-SSHOMs are composed of first-order mutants that are killed by the exact same set of test cases (nonstrict-SSHOMs are often killed by the same test cases, whereas strict-SSHOMs necessarily are killed by fewer).

Containment Relationships. In addition, we found a common containment pattern: when a (strict-)SSHOM is composed of more than two first-order mutants, it is very likely that a subset of these first-order mutants also form a (strict-)SSHOM. In other words, an *N+1 Rule*, combining a previously identified (strict-)SSHOM with one further first-order mutant is a promising strategy to identify more (strict-)SSHOMs.

Proximity. Finally, for most SSHOMs, all constituent first-order mutants are in the same class and often even in the same method, likely because first-order mutants with close proximity have higher chances of data-flow or control-flow interactions. The effect is even more pronounced for strict-SSHOMs. This stronger effect was previously conjectured though not validated [Jia and Harman, 2009].

Other. While we believe that a qualitative analysis of the mutants and their characteristics may reveal interesting insights about SSHOMs and whether they more closely mirror realistic human-made faults, such analysis goes beyond our scope of finding SSHOMs efficiently.

4.2.3 Step 3: Characteristics-Based Prioritized Search ($\texttt{search}_{pri}$)

In a final third step, we develop a new search strategy using heuristics based on characteristics found in Step 2, which will be an incomplete, but practical alternative to our $\texttt{search}_{var}$ strategy.

Our new search strategy $\texttt{search}_{pri}$ avoids the overhead of variational execution, but instead again evaluates each candidate higher-order mutant by executing the corresponding test suite, one candidate mutant at a time just like $\texttt{search}_{bf}$ and $\texttt{search}_{gen}$. Our key contribution is ordering how we explore candidate mutants to steer the search toward more likely candidates. That is, instead of a naive enumeration of all combinations ($\texttt{search}_{bf}$) or an exploration based on random seeds ($\texttt{search}_{gen}$), we prioritize based on the previously identified typical characteristics of higher-order mutants. Since characteristics for SSHOM and strict-SSHOM do not differ strongly, we develop only a single search strategy. We omit the technical detail of $\texttt{search}_{pri}$ in this proposal as it is less relevant.

Evaluation

We evaluate how *effective* our new search heuristic $\texttt{search}_{pri}$ is at finding (strict-)SSHOMs, and additionally evaluate how it *generalizes* and *scales* to much larger systems than the ones used in prior studies on SSHOMs (and used in Sec. 4.2.1). Hence, we use 3 additional subject

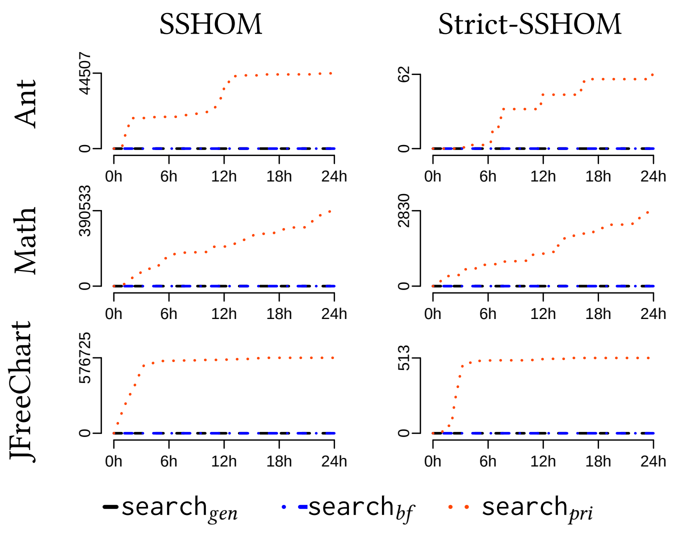


Figure 4.4: SSHOM and strict-SSHOM found over time in each subject system, averaged over 3 executions. Note that time is plotted in linear scale as SSHOMs are found consistently over time due to batching.

systems, listed in Table 4.1 (bottom), after finishing the design of our new search strategy. The new systems are significantly larger, allowing us to explore the different search strategies at a much larger (and possibly more realistic) scale.

On the small subject systems, as shown in Table 4.1 and Figure 4.3, our new search strategy `search`$_{pri}$ is often very effective, performing at least as well as and usually significantly outperforming both `search`$_{bf}$ and `search`$_{gen}$ in all subjects. In a few cases, it even outperforms `search`$_{var}$: In *Monopoly* it finds almost all higher-order mutants before variational execution finishes running the tests and in *Chess* it finds SSHOMs quickly, not limited by the effort to solve large satisfiability problems.

For the new and larger systems, our results shown in Table 4.1 and Figure 4.4 show that the baseline approaches perform very poorly at this scale. Without being informed by SSHOM characteristics the search in this vast space (e.g., 5 billion candidate combinations of mutation pairs in Math) these approaches find rarely any SSHOMs even when run for a long time. In contrast, `search`$_{pri}$ finds a significant number of (strict-)SSHOMs in each of these systems: Within 24 hours it explores most batches (91 % of all packages) and has a very high hit rate for finding actual SSHOMs among the tested candidates ($60.9\,\%$ in *Math*, $29.4\,\%$ in *Ant*, and $77.8\,\%$ in *JFreeChart*).

We conclude that `search`$_{pri}$ is an effective search strategy that scales to large systems and generalizes beyond systems from which the characteristics have been collected. While we cannot assess how many SSHOMs we are missing, our strategy is effective at finding a very large number of them in a short amount of time.

Conclusions

This chapter shows a promising application of variational execution to higher-order mutation testing. By encoding the search for SSHOMs as a `Boolean` satisfiability problem, our variational execution approach can, for the first time, identify a complete set of SSHOMs. Our evaluation results of `search`$_{var}$ give us confidence that a similar encoding approach for finding multi-edit patches is promising in automatic program repair, about which we will discuss in the next chapter. Moreover, we show that by studying the complete set of SSHOMs found by variational execution, we can learn useful insights that inspire new metaheuristics search strategies. The interesting and effective characteristics we identified for SSHOMs motivate us to study characteristics of multi-edit patches.

Chapter 5

Automatic Program Repair

Chapter 4 demonstrates that variational execution is useful in finding SSHOMs. In this chapter, we switch focus to another search problem of speculative variations. Automatic program repair has gained a lot of attention in recent years [Monperrus, 2018]. One of the most influential search-based approaches is GenProg [Weimer et al., 2009], a technique that uses genetic programming to search for a patch that passes all provided test cases. We suggest applying variational execution to such search-based approaches.

In the context of automatic program repair, speculative variants are small changes to the program under patch, and interactions are valuable combinations of these small changes that can fix the underlying bug. At the time of writing this proposal, this work is at an early stage. In this chapter, we discuss our goals and the proposed approach.

5.1 Goals

We apply variational execution to systematically explore interactions of small patch candidates, with the following three goals.

Identify Multi-Edit Patches. Despite over 10 years of active research, existing search-based automatic program repair techniques can rarely generate multi-edit patches, bug fixes that require changing multiple locations of the program. Recent empirical evidence shows that more than 70 % of faults in real-world programs are caused by buggy code in more than 2 locations [Zhong and Su, 2015], indicating that generating multi-edit patches is a promising avenue. Similar to finding SSHOMs, as discussed in Chapter 4, the main challenge of generating multi-edit patches is the lack of efficient way to navigate a exponentially large search space. Based on the key criteria discussed in Section 2.4, we argue that variational execution is a promising technique for tackling the large search space in a systematic way.

Improve Patch Quality. Our initial work shows that we can often find hundreds of solutions, combinations of small patch candidates that fix the underlying bug. The next question is which of them can better simulate what human developers would do. Recent work on patch quality distinguishes two kinds of patches—*plausible* patches that can pass all the given

```java
public Complex divide(Complex divisor) {
  //..
  if (change1) return NaN;  // actual fix
  else return isZero ? NaN : INF; // buggy, see MATH-657
  //..
}
public Complex divide(double divisor) {
  //..
  if (change2) return NaN;  // actual fix
  else return isZero ? NaN : INF; // buggy, see MATH-657
  //..
}
```

Listing 5.1: A multi-edit patch example from the Defects4J dataset

test cases, and *correct* patches that are verified by human developers. Wen et al. [2018] recently propose a ranking of patches based on contextual information in the source code. With variational execution, we have the unique opportunity to rank patches based on runtime information.

Improve Existing Search-Based Approaches. Search-based approaches like GenProg does explore multi-edit patches, but in a less systematic way due to ineffective fitness functions. Using variational execution, we can reveal dynamic information about how small patch candidates interact to fix a bug. By identifying interaction patterns of small changes, we can improve the fitness functions of existing approaches, similar to the way we study characteristics of SSHOMs to inform a metaheuristics search in Chapter 4.

5.2 Preliminary Work

Similar to Section 4.2.1, our idea is to encode multiple patch candidates as boolean options in the program, and then evaluate them and their combinations altogether by executing the test suite *once* with variational execution. The code snippet in Listing 5.1 illustrates how variational execution can inform generation of multi-edit patches. This example is extracted from a real bug fix from the Defects4J dataset [Just et al., 2014a]. Two changes are required to pass all the test cases: `change1` is necessary to pass three failing test cases, and `change2` is required to pass one of the failing test cases. Traditional search-based approaches might miss `change2` because applying it alone does not pass any new failing test cases. The root cause to this problem is that fitness function defined as the number of passing test cases is a poor proxy of partial bug fix, and thus combinations of patches are rarely explored in targeted fashion. However, with variational execution, the combination of these two changes (along with many other patch candidates) can be explored and tracked while executing the test suite. Since variational execution provides a bigger picture of patch candidates, it can inform the search of valuable combinations of changes, such as `change1` and `change2` in our example.

Since this is ongoing work, we have only preliminary evidence that shows variational execution is promising for automatic program repair. However, we argue that this direction is promising because this new application of variational execution has all key criteria discussed in Section 2.4.

Variations are patch candidates that modify a tiny part of the program. Oftentimes a lot of patch candidates are generated.
Interactions of patch candidates are important to observe because they might provide insights of synthesizing multi-edit patches, which is still an open challenge. Researchers have empirically shown that more than 70 % of bug fixes in practice require more than two repair actions Zhong and Su [2015]. Using variational execution, we could determine that multiple patch candidates are required to pass a test suite.
Sharing is very likely because of two reasons. On the one hand, patch candidates are generated independently, and thus often modified unrelated states of the program. On the other hand, the whole test suite is invoked again and again to calculate fitness, causing a lot of redundancy in executing test cases. Since each test case often tests a small part of the program, it is likely that each test case only touches on a small number of patch candidates. With the potentially abundant sharing, variational execution might be able to inspect how patches affect value differences at runtime and use the insights to guide the search of more promising patch candidates. As a side benefit, we might also gain speedup by sharing redundant executions of test cases.

We have integrated VarexC with GenProg to systematically explore interactions of hundreds of small patch candidates. Together with an implementation of bounded BDD (Section 3.2), we are using our prototype to find multi-edit patches for the IntroClass dataset, a collection of bugs in small introductory programs [Le Goues et al., 2015]. We use the Java version of IntroClass in our study [Durieux and Monperrus, [n.d.]].

Preliminary Results

Table 5.1 shows the number of bugs our approach can repair. Since our approach builds upon GenProg, we also show the results of GenProg as a baseline. The results of GenProg are obtained verbatim from a recent empirical review of repair tools [Durieux et al., 2019]. Note that the GenProg implementation from the work of Durieux et al. [2019] is different from the one we use in our approach. We obtain our implementation from the original authors of GenProg [Weimer et al., 2009]. The one used in Durieux et al. [2019] is a reimplementation of the core ideas, but might differ in minor details. For this reason, the numbers for GenProg in Table 5.1 should be taken with a grain of salt. We will collect more accurate results with our GenProg implementation in the future for a more direct comparison. Finally, we also show results for CapGen as a reference as it is the state-of-the-art [Wen et al., 2018].

In total, our approach can repair 18 out of 297 bugs. Among the 18 repaired bugs, 9 of them require at least two edits, indicating that our approach is effective at finding multi-edit patches. Moreover, for the bugs that our approach can repair, our tool can generate a bounded but complete set of usually tens of or even hundreds of plausible patches (i.e., patches that can pass all test cases). To obtain the results in Table 5.1, we set the maximum interaction degree to 3 (see Chapter 3.2), meaning that our approach can find all patches that are composed by up to 3 small edits. With the abundant set of plausible patches, we can study characteristics of

Table 5.1: Preliminary results of bugs repaired by our approach, GenProg, and CapGen on the IntroClassJava dataset. The numbers in parentheses denote the overlapping with our approach.

Project	#Bugs	Proposed Approach	GenProg	CapGen
Median	57	7	1 (1)	8 (2)
Smallest	52	11	0 (0)	11 (2)
Grade	89	0	0 (0)	3 (0)
Digits	75	0	3 (0)	3 (0)
Checksum	11	0	0 (0)	0 (0)
Syllables	13	0	0 (0)	0 (0)
Total	297	18	4 (1)	25 (4)

multi-edit patches to inspire new search strategies, similar to the way we study characteristics of SSHOMs (Chapter 4.2.2 and 4.2.3).

Comparing to GenProg, our approach can repair 14 more bugs.[1] In theory, the bugs that our approach can repair should be a superset of the bugs repaired by GenProg if the same set of small edits are used. In contrast, bugs that our approach can repair with one small edit might not be found by GenProg, depending on the stochastic genetic algorithm and the quality of fitness function. Hence, the complete set of patches we identify with variational execution can be used to evaluate the effectiveness of metaheuristic search techniques. Table 5.1 shows little overlapping between our approach and GenProg, likely due to the differences in the GenProg implementations mentioned above.

Comparing to CapGen, the state-of-the-art approach that has been evaluated on IntroClassJava [Wen et al., 2018], our approach can fix 7 fewer bugs. Unlike the comparison with GenProg, where our approach uses the same set of small edits as GenProg, CapGen uses a very different set of small edits, which is one of the main contributions and likely the main reason CapGen can repair more bugs than our approach. A further manual investigation into the bugs that are repaired exclusively by CapGen confirms this hypothesis. For example, several bugs are repaired by swapping variable names at the expression level, while GenProg mutates programs at the statement level. A little overlapping between our approach and CapGen indicates the potential of further improvement in our approach, if we adopt the mutation operators used in CapGen.

5.3 Research Plan

Since this work is at an early stage, we outline our plan for the next steps and evaluation in this section. The actual steps might vary, especially if new challenges are discovered later. However, we have confidence in our plan, based on recent experiences with higher-order mutation testing (Chapter 4).

Firstly, we plan to extend our analysis to the more practical Defects4J dataset [Just et al., 2014a], which is commonly used in automatic program repair research [Wen et al., 2018;

[1]Again, the raw numbers might not be reliable due to different implementations of GenProg.

Durieux et al., 2019]. We have finished a large portion of the planned implementation by integrating VarexC with GenProg and implementing a bounded BDD library. The remaining challenges are mostly extending VarexC to support native methods that are used in programs of Defects4J. A more detailed discussion of extending VarexC can be found in our existing work [Wong et al., 2018]. At the time of writing this proposal, our approach can analyze the Math project, which accounts for 106 out of 438 bugs in the Defects4J dataset.

Secondly, we plan to study the characteristics of identified multi-edit patches, both qualitatively and quantitatively. Similar to the way we study characteristics of SSHOMs, the main goal is to observe what makes a multi-edit patch or high-quality patch, to further inspire new ideas in automatic program repair research, such as ways to design better fitness functions. We also plan to analyze runtime information collected during variational execution for a more fine-grained observation. Based on the experiences with SSHOMs and preliminary results discussed in the previous section, we believe this step has low risks.

Finally, we plan to demonstrate the effectiveness of the identified characteristics by implementing a new metaheuristic search strategy, similar to the $\texttt{search}_{pri}$ in Chapter 4.2.3. The new search can also serve as a mitigation strategy for the first step mentioned above, in case extending VarexC to Defects4J takes too much engineering effort. That is, we can study characteristics of patches based on the data obtained from IntroClassJava or other small programs, and then use Defects4J to evaluate our new metaheuristic search strategy. Ideally, we would use Defects4J to obtain complete sets of patches for studying characteristics, as Defects4J is a better proxy of realistic bug fixes [Just et al., 2014a] than IntroClassJava. In that case, we plan to use another existing dataset to evaluate the new search, such as BugSwarm [Tomassi et al., 2019].

After performing the planned steps, we evaluate our approach from the following aspects:

- We evaluate how interaction degree affects the number of bugs we can repair and the number of patches our approach can generate. The goal of this evaluation is to have a better understanding of the configuration space.
- We compare our variational execution approach with existing approaches in terms of the number of repaired bugs, similar to Table 5.1. To isolate the effect of using variational execution, we will compare approaches that use similar mutation operators, or extend GenProg to adopt more recent mutation operators.
- We compare the execution time of our variational execution approach to estimate how practical our approach is.
- We report the identified characteristics of patches together with quantitative data that demonstrates prevalence and qualitative findings that reveal interesting insights.
- Finally, we evaluate the new characteristics-based search strategy in terms of number of bugs repaired, patch quality, and execution time. The goal is to demonstrate usefulness of our identified characteristics, and more importantly inspire new ideas for future research.

Chapter 6

Research Plan

In this chapter, we summarize the remaining steps *ordered by priority*, and estimate the time needed to finish the proposed thesis.

At the time of writing, our work on higher-order mutation testing (Chapter 4) is finished but still under review. In case the work is not accepted, we might polish our paper to improve writing, but we do not plan to conduct further research or evaluation, as we believe the existing work has sufficiently demonstrated the usefulness of variational execution in higher-order mutation testing. We estimate the potential polishing of writing takes **2 weeks**.

The major missing component of the proposed thesis is applying variational execution to automatic program repair (Chapter 5). As discussed in Chapter 5.3, there are a few remaining steps:

- Extending VarexC to repair bugs in the Defects4J dataset and run experiments to collect evaluation results. Based on the prior experiences with VarexC and the time spent on setting up Apache Math, one of the Defects4J programs, we estimate this step takes **2 months**.

- Analyzing the characteristics of the complete set of patches found by our variational execution approach. We plan to conduct a mixed method study that qualitatively observes hypotheses from a sample of patches and then quantitatively validates them in all the data. We suspect running experiments in the previous step could take nontrivial amount of time, so we plan to conduct the qualitative part incrementally while running experiments. In addition, we estimate that it takes another **1 month** to finish the study, which is roughly the time we spent on studying characteristics of SSHOMs (Chapter 4.2.2).

- Designing and evaluating a new metaheuristic search strategy using the characteristics we identified. This step involves an engineering component to design and implement the new metaheuristic search, but we suspect that we can reuse a lot of infrastructure code from the existing GenProg implementation. We estimate that we need **2 months** to finish this step.

- Finally, we reserve another **1 month** to write and submit the work to a major conference.

We have plans for a systematic research on bounded variational execution (Chapter 3.2), which includes (1) a systematic literature survey about existing work (if there is any); (2) formalizing the essential ideas of bounded BDD with necessary correctness proofs; (3) evaluating our bounded BDD implementation to demonstrate practicality; (4) publishing the work as a conference paper. As discussed, this line of work is a valuable addition to the proposed thesis, but optional as the main goal of the thesis is to explore interactions of speculative variations, for which we already implemented a well-tested prototype of bounded variational execution. We estimate that this work takes **3 months**.

Finally, we reserve **2 months** for writing the proposed thesis. While writing, we also plan to work with an undergraduate student in the summer through the Research Experiences for Undergraduates in Software Engineering (REUSE) program at CMU. We do not have a concrete plan for the REUSE project at the time of writing, as it highly depends on the skills and interests of the undergraduate student. But overall, we plan to work on some standalone ideas that can benefit the proposed thesis, such as studying other categories of higher-order mutants beyond SSHOMs, or investigating yet another new application of variational execution to solve an open challenge in search-based software engineering. Similar to the bounded variational execution project, the REUSE project is not on the critical path of this thesis.

In summary, we expect to finish the proposed thesis in **11 months**, before November 2020.

Bibliography

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7 Page 7, Page 22

Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09).* ACM, New York, NY, USAB, 113–124. https://doi.org/10.1145/1554339.1554353 Page 12

Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12).* ACM, New York, NY, USA, 165–178. https://doi.org/10.1145/2103656.2103677 Page 2, Page 3, Page 12, Page 17

Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '13).* ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2465106.2465121 Page 12, Page 17

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (May 2018), 1–39. https://doi.org/10.1145/3182657 Page 11

Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (Aug. 1986), 677–691. https://doi.org/10.1109/TC.1986.1676819 Page 22, Page 23, Page 32

Isis Cabral, Myra B. Cohen, and Gregg Rothermel. 2010. Improving the Testing and Testability of Software Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10).* Springer-Verlag, Berlin, Heidelberg, 241–255. Page 2

Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (Jan. 2003), 115–141. https://doi.org/10.1016/S1389-1286(02)00352-3 Page 1, Page 11

Deepak Chandra and Michael Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, Miami Beach, FL, USA, 463–475. https://doi.org/10.1109/ACSAC.2007.37 Page 12

Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, New York, NY, USA, 129–139. https://doi.org/10.1145/1273463.1273482 Page 2

Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. 2007. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, New York, NY, USA, 50–60. https://doi.org/10.1145/1273463.1273472 Page 13

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 748–759. https://doi.org/10.1145/2382196.2382275 Page 11

Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-Execution. In *2010 IEEE Symposium on Security and Privacy*. IEEE, Berkeley/Oakland, CA, 109–124. https://doi.org/10.1109/SP.2010.15 Page 11

Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. *arXiv:1905.11973 [cs]* (May 2019). arXiv:cs/1905.11973 Page 41, Page 43

Thomas Durieux and Martin Monperrus. [n.d.]. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. ([n. d.]), 7. Page 41

Martin Erwig and Eric Walkingshaw. 2013. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV*, Ralf Lämmel, João Saraiva, and Joost Visser (Eds.). Vol. 7680. Springer Berlin Heidelberg, Berlin, Heidelberg, 55–100. https://doi.org/10.1007/978-3-642-35992-7_2 Page 8

R. Gopinath, C. Jensen, and A. Groce. 2017. The Theory of Composite Faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 47–57. https://doi.org/10.1109/ICST.2017.12 Page 29

M. Harman, Y. Jia, and W. B. Langdon. 2010. A Manifesto for Higher Order Mutation Testing. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 80–89. https://doi.org/10.1109/ICSTW.2010.13 Page 29

Mark Harman, Yue Jia, Pedro Reales Mateo, and Macario Polo. 2014. Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from

Strongly Subsuming Higher Order Mutation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering - ASE '14*. ACM Press, Vasteras, Sweden, 397–408. https://doi.org/10.1145/2642937.2643008 Page 27, Page 29, Page 33

Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-Based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1 (Nov. 2012), 1–61. https://doi.org/10.1145/2379776.2379787 Page 2, Page 3

Klaus Havelund and Thomas Pressburger. 2000. Model Checking JAVA Programs Using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)* 2, 4 (March 2000), 366–381. https://doi.org/10.1007/s100090050043 Page 18

Petr Hosek and Cristian Cadar. 2013. Safe Software Updates via Multi-Version Execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 612–621. Page 11

Hao Huang. 2019. Induced Subgraphs of Hypercubes and a Proof of the Sensitivity Conjecture. *arXiv:1907.00847 [cs, math]* (Aug. 2019). arXiv:cs, math/1907.00847 Page 24

Yue Jia. [n.d.]. *Higher Order Mutation Testing*. Ph.D. Dissertation. Page 29

Yue Jia and Mark Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, Beijing, 249–258. https://doi.org/10.1109/SCAM.2008.36 Page 29

Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Information and Software Technology* 51, 10 (Oct. 2009), 1379–1393. https://doi.org/10.1016/j.infsof.2009.04.016 Page 1, Page 27, Page 29, Page 30, Page 32, Page 35

René Just, Darioush Jalali, and Michael D. Ernst. 2014a. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055 Page 40, Page 42, Page 43

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014b. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929 Page 27, Page 29

René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 284–294. https://doi.org/10.1145/3092703.3092732 Page 29

Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-Aware Testing. In *Proceedings of* *the 4th International Workshop on Feature-Oriented Software Development - FOSD '12*. ACM Press, Dresden, Germany, 1–8. https://doi.org/10.1145/2377816.2377817 Page 3, Page 12

Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, Dallas, TX, USA, 221–230. https://doi.org/10.1109/ISSRE.2012.23 Page 12

M. Kintis, M. Papadakis, and N. Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *2010 Asia Pacific Software Engineering Conference*. 300–309. https://doi.org/10.1109/APSEC.2010.42 Page 29

Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-Cloaking Internet Malware. In *2012 IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 443–457. https://doi.org/10.1109/SP.2012.48 Page 11

B. Kurtz, P. Ammann, and J. Offutt. 2015. Static Analysis of Mutant Subsumption. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–10. https://doi.org/10.1109/ICSTW.2015.7107454 Page 29

Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the Validity of Selective Mutation with Dominator Mutants. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 571–582. https://doi.org/10.1145/2950290.2950322 Page 29, Page 34

Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16*. ACM Press, Atlanta, Georgia, USA, 503–515. https://doi.org/10.1145/2872362.2872395 Page 11, Page 12

William B. Langdon, Mark Harman, and Yue Jia. 2010. Efficient Multi-Objective Higher Order Mutation Testing with Genetic Programming. *Journal of Systems and Software* 83, 12 (Dec. 2010), 2416–2430. https://doi.org/10.1016/j.jss.2010.07.027 Page 29

Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec. 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513 Page 41

L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (Jan. 2014), 23–42. https://doi.org/10.1109/TSE.2013.44 Page 29

P. Reales Mateo, M. Polo Usaola, and J. L. Fernández Alemán. 2013. Validating Second-Order Mutation at System Level. *IEEE Transactions on Software Engineering* 39, 4 (April 2013), 570–587. https://doi.org/10.1109/TSE.2012.39 Page 29

Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. 2018. Understanding Differences among Executions with Variational Traces. *arXiv:1807.03837 [cs]* (July 2018). arXiv:cs/1807.03837 Page 3, Page 10, Page 12

Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/2970276.2970322 Page 1, Page 2, Page 3, Page 9, Page 11, Page 12, Page 13, Page 14, Page 17, Page 18, Page 21, Page 22, Page 33

Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 2016. How Does the Degree of Variability Affect Bug Finding?. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 679–690. https://doi.org/10.1145/2884781.2884831 Page 1

Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan. 2018), 1–24. https://doi.org/10.1145/3105906 Page 1, Page 2, Page 22, Page 39

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 907–918. https://doi.org/10.1145/2568225.2568300 Page 2, Page 3, Page 12, Page 17, Page 33

Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. 2008. Feature Interaction: The Security Threat from within Software Systems. *Progress in Informatics* 5 (March 2008), 75. https://doi.org/10.2201/NiiPi.2008.5.8 Page 1, Page 11

Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2 (Feb. 2011), 11:1–11:29. https://doi.org/10.1145/1883612.1883618 Page 2

A. Jefferson Offutt. 1992. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology* 1, 1 (Jan. 1992), 5–20. https://doi.org/10.1145/125489.125473 Page 29

Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. 2017. Subtle Higher Order Mutants. *Inf. Softw. Technol.* 81, C (Jan. 2017), 3–18. https://doi.org/10.1016/j.infsof.2016.01.016 Page 29

Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015 Page 1, Page 2, Page 27

M. Papadakis and N. Malevris. 2010. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 90–99. https://doi.org/10.1109/ICSTW.2010.50 Page 29

D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623 Page 3

Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. ACM, New York, NY, USA, 163–171. https://doi.org/10.1145/3183519.3183521 Page 29

G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 47–53. https://doi.org/10.1109/ICSTW.2018.00027 Page 29

Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. 2009. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Softw. Test. Verif. Reliab.* 19, 2 (June 2009), 111–131. https://doi.org/10.1002/stvr.v19:2 Page 29

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 445–454. https://doi.org/10.1145/1806799.1806864 Page 2, Page 12, Page 14, Page 21, Page 22

Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. 2018. Faceted Secure Multi Execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*. ACM Press, Toronto, Canada, 1617–1634. https://doi.org/10.1145/3243734.3243806 Page 12, Page 17

Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635*. Springer-Verlag New York, Inc., New York, NY, USA, 3–23. https://doi.org/10.1007/978-3-662-49635-0_1 Page 12, Page 17

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 842–853. https://doi.org/10.1145/2786805.2786830 Page 2, Page 11, Page 13

Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. 2018. VarXplorer: Lightweight Process for Dynamic Analysis of Feature Interactions. In *Proceedings of the 12th International Workshop on Variability Modelling* *of Software-Intensive Systems - VAMOS 2018*. ACM Press, Madrid, Spain, 59–66. https://doi.org/10.1145/3168365.3168376 Page 12

Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/1294261.1294284 Page 11

William N. Sumner, Tao Bao, Xiangyu Zhang, and Sunil Prabhakar. 2011. Coalescing Executions for Fast Uncertainty Analysis. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 581–590. https://doi.org/10.1145/1985793.1985872 Page 13

William N. Sumner and Xiangyu Zhang. 2013. Comparative Causality: Explaining the Differences Between Executions. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 272–281. Page 11, Page 12

David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-Gonzalez. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 339–349. https://doi.org/10.1109/ICSE.2019.00048 Page 43

Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. 2009. Efficient Online Validation with Delta Execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/1508244.1508267 Page 13

Alexander von Rhein, Sven Apel, and Franco Raimondi. 2011. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *Proc. Java Pathfinder Workshop*. 82. Page 2, Page 13

Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster Mutation Analysis via Equivalence Modulo States. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/3092703.3092714 Page 13

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536 Page 39, Page 41

Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. ACM Press, Gothenburg, Sweden, 1–11. https://doi.org/10.1145/3180155.3180233 Page 40, Page 41, Page 42

Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution with Transparent Bytecode Transformation. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 117:1–117:30. https://doi.org/10.1145/3276487 Page 2, Page 17, Page 18, Page 20, Page 21, Page 33, Page 43

Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 631–647. https://doi.org/10.1145/2908080.2908098 Page 12

Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '02/FSE-10)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053 Page 11, Page 12

Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 913–923. Page 1, Page 29, Page 39, Page 41