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

**Chu-Pan Wong**

January 2021

CMU-ISR-21-100

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Christian Kästner (Chair)
Claire Le Goues
Heather Miller
Abhik Roychoudhury (National University of Singapore)

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.*



This research was supported in part by the NSF (awards 1318808, 1552944, 1717022), AFRL, and DARPA (FA8750-16-2-0042). The views continaed in this document are those of the author, and should not be interpreted as representing the offcial policies, either expressed or implied, of any sponsoring institution, or any other entity.

**Keywords:** Dynamic Analysis, Variational Execution, Configurable System, Automatic Program Repair, Mutation Testing, Bytecode Transformation

## Abstract

Variations are ubiquitous in software. Some variations are *intentionally* introduced, e.g., to provide extra functionalities or tweak certain program behavior, while some variations are *speculatively* generated to achieve certain search goals, such as 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. There is increasing evidence showing the importance of studying 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 large search spaces of interactions among variations.

Among existing approaches that study interactions of *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 an improved implementation of 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, using variational execution and SAT solving techniques to iterate all solutions efficiently. For automatic program repair, our approach can systematically explore the search space to find high-quality multi-edit patches. For higher-order mutation testing, our approach can find a *complete* set of solutions with regard to a given search space, enabling further study on their characteristics to understand their nature and learn useful insights to inspire new ideas, such as lightweight but more effective heuristics-based search strategies.

# Acknowledgments

This dissertation would not exist without the help and support from my advisor, my collaborators, my family, and my friends.

First and foremost, I would like to thank my advisor Christian Kästner, who had offered me tremendous support throughout my PhD. I was lucky to have Christian as my guide on my research journey. Through Christian, I learnt coding, writing, speaking, teaching, collaborating, cooking, and a bit of juggling. I admire his passion and dedication to research. He has always been my most reliable source of advice and solace during the most difficult times. Christian has been and will always be my role model as a researcher.

I would like to thank my committee members Claire Le Goues, Heather Miller, and Abhik Roychoudhury. I greatly appreciate their time and feedback. Their input to my research and dissertation is invaluable.

I am grateful to my undergrad mentors Yingfei Xiong and Lu Zhang at Peking University. They gave me the first taste of research. Without their support and encouragement, I could not have started my research journey at Carnegie Mellon.

Special thanks to my closest collaborator Jens Meinicke, who has always been helpful and supportive. I appreciate the countless hours we have spent together on various research projects since the first day of my PhD.

I would like to thank all other collaborators Leo Chen, João P. Diniz, Eduardo Figueiredo, Dan Hao, Lukas Lazarek, David Lo, Hong Mei, Meng Meng, Gunter Saake, Mauricio Soto, Thomas Thüm, Ferdian Thung, Bogdan Vasilescu, Eric Walkingshaw and Hongyu Zhang, for their guidance and inspiration in research. It has been my pleasure to work with these wonderful researchers, and I wish we can extend our collaboration in the future.

None of my accomplishment matters without my parents Kam-Fai Wong and Hongmei Zu. I am forever indebted to them for their love, care, and education that have made me who I am. They have always been selflessly supporting me to challenge myself, to learn from failure, and to be grateful for what life has given me.

Special thanks to my dear friend Shurui Zhou for always being there when I need support. I wish you all the best for future endeavors at University of Toronto. Thanks to other members in our research group Gabriel Ferreria, Pooyan Jamshidi, and Miguel Velez, for all the thought-provoking discussions and feedback.

I would like to also thank my friends and colleagues at Carnegie Mellon. Thanks to Tobias Dürschmid, David Widder, and Roykrong Sukkerd for all the fun at board game parties. Thanks to Iain Cruickshank and Matt Benigni for being the nicest officemates. Thanks to Kyle Liang for the grocery adventures. Thanks to Wode Ni for the fun on pool tables. Thanks to Connie Herold, Jamie Lou Hagerty, Jennifer Cooper, Ryan Johnson, and Emanuel Bowes for all the prompt assistance. And finally, thanks to all other researchers and friends at Carnegie Mellon for sharing their knowledge and expertise with me.

This journey has been exciting and fulfilling thanks to all of you.

# 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 [113]; and mutation testing approaches mutate an existing program to create different mutants to assess the quality of the existing test suite [125].

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 [21, 118]. Conflicts among variations arise when one variation interferes with another in an unintended way, which is difficult to foresee even in small programs [111]. On the other hand, interactions among *speculatively* created variations are also of interests to researchers. A recent study by Zhong and Su [175] 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 [56] 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 [110].
- 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 [7, 117]. 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 [113]. Mutation testing approaches typically create first-order mutants, mutated programs that have exactly one small change [125]. 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 [48].

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 [20, 24, 119]. On the heavyweight side, there are verification approaches that use model checking or symbolic execution to statically analyze all possible interactions [134, 143, 157]. 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 [7, 110, 117, 166]. 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 [66, 117], understanding feature interactions and configuration faults [109, 110], and monitoring information flow of sensitive data [7]. 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 mutations 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 [48].

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 [48]. 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 [127]. 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), 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). 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 tackle the large search spaces efficiently, capturing interesting interactions systematically and effectively.

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

- Drawing inspiration from intentional variations, we analyze two successful applications of variational executions—testing highly configurable systems and tracking information flow—to identify ingredients of promising applications. These ingredients serve as a guideline for future research on applying variational execution (Chapter 2).
- 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 (Chapter 3).
- 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 (Chapter 3).
- We use variational execution to find strongly subsuming higher-order mutants (SSHOM), a special kind of higher-order mutant that can potentially 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 metaheuristics search in terms of time spent and SSHOMs found (Chapter 4).
- 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 metaheuristics search (Chapter 4).
- Using a similar recipe, we apply variational execution to automatic program repair. Comparing to prior work, our approach can explore a vast search space of diverse fixing ingredients in a systematic way. A such like this can shed light on several open challenges in automatic program repair, such as generating general patches, high-quality patches and multi-edit patches (Chapter 5).
- We objectively evaluate our approach on two widely used datasets. The results indicate that variational execution as a search strategy can effectively navigate the search space of diverse fixing ingredients to fix buggy programs. Moreover, the systematic search of variational execution enables us to explore interesting interactions among fixing ingredients, yielding many multi-edit patches and high-quality patches (Chapter 5).

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.

## 1.4 Outline

The remainder of the dissertation 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.
- 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 showcases another application of variational execution, using it as a search strategy to navigate large search spaces in automatic program repair.
- In Chapter 6, we conclude the dissertation, summarize potential impact, and briefly outline future work.

# Chapter 2

# Criteria of Applying Variational Execution

This chapter has two parts. First, we provide the essential background on variational execution, specifically why it is effective in exploring interactions of variations. Although the conceptual ideas of variational execution are not new, existing applications span across multiple domains and there is lack of conceptual criteria for gauging potential applications. To that end, in the second part of this chapter, we analyze two important and successful applications of variational execution, with the goal to distill key ingredients for future applications.

This chapter shares material with the following publications [165, 166]:

- Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster variational execution with transparent bytecode transformation. Proc. ACM Program. Lang. 2, OOPSLA, Article 117 (November 2018), 30 pages.
- Chu-Pan Wong, Jens Meinicke, and Christian Kästner. 2018. Beyond testing configurable systems: applying variational execution to automatic program repair and higher order mutation testing. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 749–753.

## 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 behavior 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 [5]. 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 value* and *variability context*.

**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* [37]. 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 efficiently.

**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 operation only needs to be performed 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 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. [109] 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 $\alpha$, $\beta$, $\gamma$ 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

[1] https://wordpress.org

Listing 1: Original version

```
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 }
```

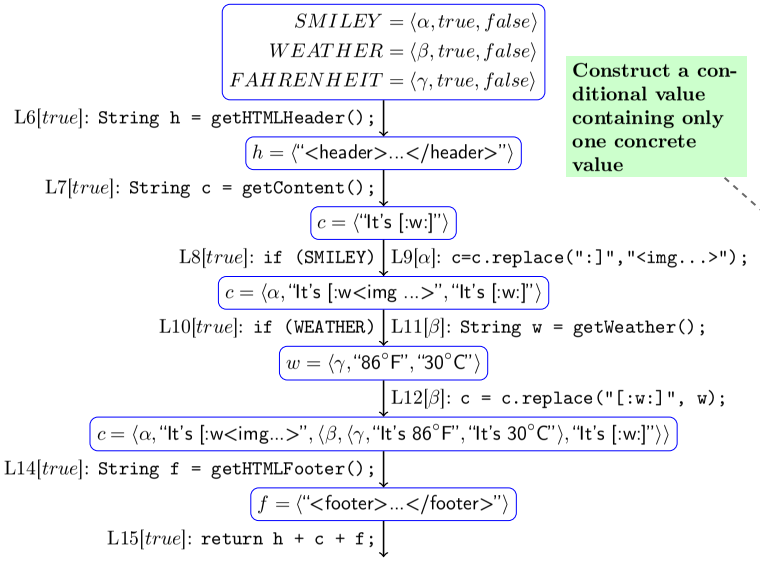
Construct a conditional value containing two alternative values

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

```
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;
}
```

Figure 2.1: Running example modeled after WordPress [110]. 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 instead of bytecode for better readability.

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

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. [110] 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 [10, 143]. 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.

Finally, variational execution is fundamentally different from traditional approaches of multi-execution [27, 29, 50, 76, 147] and delta debugging [82, 149, 173] 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 [21, 118], 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. [117] 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. [110] and Kim et al. [69] executed Java programs with configuration parameters 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 [66, 69, 117]. Soares et al. [146] 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 [109], in line with delta debugging [82, 149, 173]. Reisner et al. [134] 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 [110].

### 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, variations are 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 [6, 22]. Austin and Flanagan [7] 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 [7]). 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 [8, 140, 141, 171].

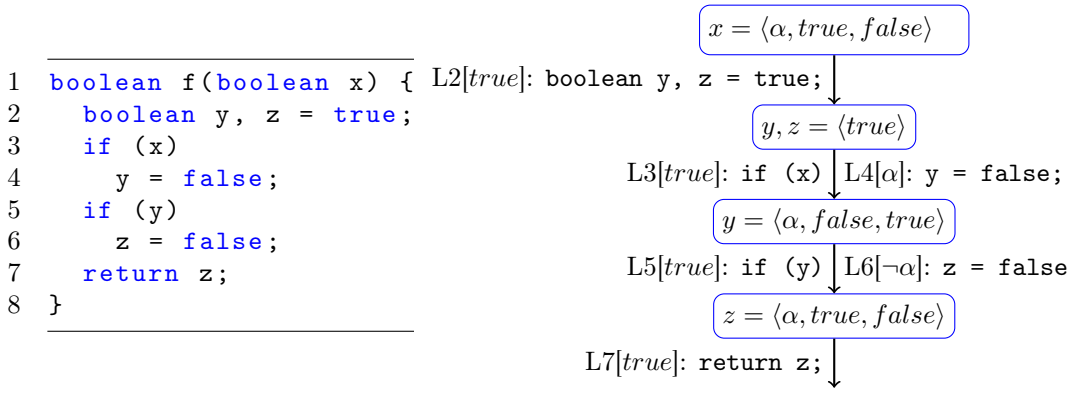



Figure 2.2: An example illustrating how variational execution can be used to handle implicit information flow.

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.

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 $\alpha$ 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. [148] shares similarities among executions of simulation workloads and computes with several values in parallel. Wang et al. [159] 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. [155] 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 explore interesting interactions.

Similar ideas of sharing computations can be found also in approaches for model checking and symbolic execution [25, 143, 157], 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. [110] 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 [165]. To predict applicability of variational execution in new domains, it is useful to elicit the key characteristics of the existing successful applications.

To that end, we analyzed how variational execution has been successfully applied to two important domains—configuration testing and information flow tracking. The analysis resulted in 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 [110, 134]. 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 [110, 134]. 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.

These criteria can be useful not only for identifying similar applications of intentional variations, but also for predicting new applications of speculative variations, which is the main goal of this thesis as has been noted. Using these criteria, we briefly discuss the potential of applying variational execution to two important problems of speculative variations—higher-order mutation and automatic program repair. We will discuss these applications in greater length in Chapter 4 and Chapter 5.

**Higher-order mutation testing** has all the key enablers of variational execution. *Variations* are used to encode mutations, so we can easily get many variations by generating mutations randomly. *Sharing* is very likely due to the random generation of mutations and local effect of many mutations. With variational execution, we can execute the test suite once and observe the effect of all mutations, avoiding repeated executions of the same test suite. *Interactions* are interesting to inspect because we could detect valuable higher-order mutants that are much harder to kill than its constituent first-order mutants.

**Automatic program repair** is also a promising application. *Variations* are fixing ingredients that modify a tiny part of the program. Oftentimes many fixing ingredients are generated to increase the likelihood of finding a patch. *Interactions* of fixing ingredients are important to observe because they might provide insights of synthesizing multi-edit patches, which is still an open challenge in the field. *Sharing* is very likely because of two reasons. On the one hand, fixing ingredients are generated independently, and thus often modify unrelated states of the program. On the other hand, tests are invoked again and again to calculate fitness, causing a lot of redundancy in test executions. As a side benefit, we can also inspect how fixing ingredients affect program state at runtime and use the insights to guide the search of high-quality patches.

## 2.5 Summary

In this chapter, we introduced the key ideas of variational execution, analyzed existing applications, and more importantly, elicited three important criteria for gauging the potential of applying variational execution to new domains. New domains that concern speculative variations are emerging, but understanding interactions of speculative variations remain challenging. Using the three key criteria, we show that variational execution is useful for higher-order mutation testing (Chapter 4) and automatic program repair (Chapter 5). As discussed in 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 [8, 110, 117], but none of them is scalable or extensible enough for analyzing speculative variations. 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.

This chapter shares material with the following publication [166]:

- Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster variational execution with transparent bytecode transformation. Proc. ACM Program. Lang. 2, OOPSLA, Article 117 (November 2018), 30 pages.

## 3.1 Faster Variational Execution

Existing implementations of variational execution rely on either *manual modification to the source code* [8, 140, 141] or *modification to the language interpreter* [110, 117].

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 [8, 140, 141]. 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 [7, 110, 117]. 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 [49], which however provides incomplete language support and often mediocre performance. For example, VarexJ [110], the state-of-the-art variational execution engine for Java, is implemented on top of Java PathFinder's (JPF) interpreter for Java bytecode [49]. 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.

Transformations are nontrivial and not always local. While many bytecode instructions can be transformed in isolation, encoding conditional control flow in a commodity JVM requires careful encoding, such that both branches of control-flow decisions can be executed in different variability contexts before subsequent computations are merged again to maximize sharing overall. In additional, data-flow analyses are required to handle values on the operand stack between blocks and object initialization sequences for variational execution. Finally, we perform additional optimizations to statically pinpoint instructions that do not need to be transformed, if they are guaranteed to be not related to variations in the program.

We formally prove that our transformation of control flow is correct, statically guarantee optimal sharing for a large subset of possible control-flow graphs. Additionally, we empirically evaluate performance, comparing execution time and memory consumption on seven highly configurable systems against VarexJ, the state-of-the-art variational execution implementation. The results show that our approach is 2 to 46 times faster than VarexJ while using 75 percent less memory. The performance results also indicate that our approach is efficient for analyzing highly configurable systems in practice.

We summarize our contributions as follow:

- We propose a novel strategy for variational execution using automatic bytecode transformation, without any manual modifications to the source code or to the language interpreter.
- We prove that our automatic transformation of bytecode is correct for all control-flow graphs and optimal with regard to sharing for a large subset.
- We propose further optimizations by performing data-flow analysis and using specialized data structures.
- We implement a bytecode transformation tool that covers nearly the entire instruction set of the Java language, with minor exceptions that we explain in Section 3.5. The

transformed bytecode is portable to any implementation of the JVM specification.
- An empirical evaluation with 7 subject systems showing that our approach is up to 46 times faster while saving up to 75 percent memory when compared to the state of the art. In addition to statically guaranteeing optimal sharing for 89.7 percent of methods, our approach achieves optimal sharing at runtime for 99.8 percent of all other method executions.

We hope that the way we transform bytecode can inspire more efficient implementation of similar techniques such as symbolic execution. Although we focus on Java bytecode in this work, we can potentially generalize the core ideas to other programming languages and other analyses, by performing a similar transformation at the well-defined intermediate representation form of existing compiler frameworks like LLVM.

## 3.2 Motivation: A Manual Rewrite

To motivate transforming bytecode automatically, we illustrate how the source code of our earlier WordPress example in Chapter 2 can be manually rewritten from Listing 1 to Listing 2 in Figure 2.1 on page 10. 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 on page 10) 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 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. As we will see later in Section 3.5, our transformation is also able to automatically decide which parts of a program need to be transformed, as it is likely that some parts are not related to variations, such as the code before the first `if` statement (Line 7) and the code after the second `if` statement (Line 14) in Listing 2.

## 3.3 Bytecode Transformation

We discuss our transformation in two steps. First, in this section, we discuss how to transform all instructions that are executed in a given variability context. The transformation of control flow, which may change variability context, is nontrivial and orthogonal, so we discuss it second in Section 3.4. We describe transformations for similar instructions together, following the grouping of the JVM specification [93].

In a nutshell, we transform each bytecode instruction of the original program into a sequence of bytecode instructions. Ideally, the transformation of most 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 used internally in the JVM for exchanging data between instructions. Some instructions load values (e.g., constants or values from local variables or fields) onto the operand stack, while other instructions take values from the operand stack and operate on them. Results might be pushed back onto the operand stack as a result of an operation. The operand stack is also used to prepare parameters to be passed to method invocations and to receive return values.

To assist local transformation of individual instructions, we introduce several transformation invariants:

**Invariant 1** All local variables and fields store conditional values.
**Invariant 2** All values on the operand stack are conditional values.
**Invariant 3** All methods take conditional values as parameters and return conditional values.

We ensure that these invariants hold *before and after* the execution of each transformed bytecode sequence. They help us establish a common ground about the environment, enabling concise transformations of most instructions. In addition, we assume that each instruction is executed in a local variability context. We will explain how variability contexts are propagated and changed as part of our discussion of control flow in Section 3.4.

### 3.3.1 Basic Lifting

To achieve our invariants, we change all parameters and local variables in a method frame to the `V` type to store conditional values. Primitive types are boxed in the process.

**Load and Store Instructions.** Load and store instructions transfer values between the local variables and the operand stack. Since we assume local variables and stack values to represent conditional values (**Invariant 1**, **Invariant 2**), we can directly load them with the `aload` instruction (replacing load instructions for primitive types if needed). Store instructions require more attention, because they may be executed under a restricted variability context, in which case not all values shall be overwritten. For example, suppose we have $x = 1$ under context `true`, but store 2 to $x$ under context `A`, then $x$ stores the conditional value $\langle A, 2, 1\rangle$ instead of 2. To this end, we always create a new conditional value, compressing the updated values under the current context with possibly unaffected old values. As an example, consider the `V` constructor call when `c` is updated in Line 38–40 of Listing 2 in Figure 2.1 on page 10.

**Arithmetic and Type Conversion Instructions.** Arithmetic and type conversion instructions compute a result based on one or two values from the operand stack, and then push the result back on the operand stack. For example, the `iadd` instruction takes two *int* values from the stack, adds them together and pushes the result back. Given **Invariant 2**, we need to pop and push conditional values. We achieve this by invoking `smap` with the current variability context on the stack's conditional values, performing the original arithmetic or type conversion operation on each alternative concrete value. For operations on two conditional values, we combine `sflatMap` and `smap` to compute results for all possible combinations. For example, the original floating point calculation in Line 21 of Figure 2.1 on page 10 is transformed to a `smap` call in Line 63.

**Operand Stack Management Instructions.** Operand stack management instructions directly manipulate entries on the operand stack, such as `pop` for discarding the top value, and `swap` for swapping the top two values. They work the same for conditional values and concrete values, and therefore do not need to be transformed. A technical subtlety in Java is that some primitive values (e.g., long, double) are represented by two 32-bit values on the stack, but only by a single reference value for a conditional value; here we adjust stack operations accordingly.

### 3.3.2 Method Invocation and Return

Method invocations pass the top stack values as arguments to the method and push the method's result back to the stack. Non-static methods also take their receiver from the stack. Since method arguments and return types are conditional values, just as stack values (**Invariant 2**, **Invariant 3**), they can be passed along directly. If a method call has multiple receiver objects, we call the method for each of them in the corresponding variability context and merge results using a `sflatMap` call.

Special handling is required though in cases in which **Invariant 3** does not hold for the target. Ideally, all classes and all methods in variational execution should be transformed, but this

is not always possible in practice because of the environment barrier. At some point, variational programs may need to interact with an environment that does not know about variational values and variability contexts. The environment barrier can be at different places, depending on how the system is implemented (e.g., between user code and library code, between Java code and native code, between the program and the operating system or network), but can never be avoided entirely. When hitting the environment barrier, we have three options:

- **Multiple Invocations.** For *side effect free* methods, we can invoke the target method multiple times for each feasible combinations of concrete argument values, merging the results into a single conditional value. Since the method is side effect free, invoking it repeatedly with different arguments does not change the program states, it just forgoes potential sharing.
- **Model Classes.** We can always provide variational models for the environment, for example, replacing all reads and writes to a file with a special implementation that can store alternative file context under different contexts. Such model classes are common in model checking and symbolic execution [25, 143, 157] and have been explored in variants of variational execution for database storage [171]. Model classes can also be used to provide more efficient variational implementations for classes than would be achieved with our automated transformation, as we will discuss in Section 3.5.
- **Abort.** Finally, we can execute the program but abort execution when we reach the environment barrier at runtime. This way, we can still support executions that do not cross the barrier, even though the source code refers to nonvariational methods. Furthermore, we can allow calls to nonvariational methods during the execution when they are shared by all configurations (with variability context *true*) in which all parameters have only a single concrete value.

In our approach, we transform all methods possible, including libraries, to push the environment barrier as far outside as possible. In the JVM, the environment barrier often manifests as native methods, i.e., methods that are hard-coded in the JVM in other programming languages such as C and C++. We maintain a list of model classes and side-effect free methods that are automatically applied when encountered. For all remaining calls to nonvariational code, we issue warnings during transformation and abort the execution at runtime when invoked. We then manually and incrementally inspect aborts in our executions and mark methods as side-effect free or develop model classes as needed. In fact, so far, we needed to implement model classes only for a small number of classes. We have not yet encountered executions that heavily rely on variational interactions with the environment and thus require additional model classes.

Return instructions are more straightforward to transform than method invocation instructions. To not prematurely end the execution of a method at a return instruction, we rewrite the method to use a single return instruction at the end of the method. If the method being transformed has more than one return instructions, we rewrite all of them to jump to a single return at the end of the transformed method. If necessary, we store the values of different original return instructions in a variable. Technically, we again replace all non-void returns by a single `areturn` instruction, returning a reference to the resulting conditional value. For example, see how Line 21 and 23 are transformed to Lines 62, 67 and 70 in Figure 2.1 on page 10.

### 3.3.3 Using Objects

In the JVM, both *class instances* and *arrays* are objects, but the JVM creates and manipulates class instances and arrays using distinct sets of instructions. This section presents our transformation of them respectively.

**Class Instances.** We transform all fields of a class instance to have the conditional value type. The key idea is to maximize sharing of data across similar class instances. If two instances of the same class only differ in one field, we represent the difference in a conditional value for that field, rather than as a conditional reference to two copies of the object. This design stores variability as local as possible to avoid redundancy in memory and in computations [110]. As fields store conditional values (**Invariant 1**), reads and writes to fields work just as loads and stores to local variables.

A technical challenge to independent transformation of bytecode instructions arises for the `new` instruction used to instantiate classes and push them to the operand stack. The challenge is that the `new` instruction creates an uninitialized object that cannot be passed as a reference for safety reasons until the object's constructor is invoked on it, and thus cannot be wrapped in a `V` type as needed for **Invariant 2**. Instead, we treat `new` and the subsequent initialization sequence as *one bytecode instruction* for our transformation. Whenever we encounter a `new` instruction, we use a data-flow analysis to identify the relevant following initialization sequence, re-arranging the original bytecode if necessary to separate object initialization from other instructions (e.g., instructions to compute constructor parameters).

**Arrays.** For a given array, we transform it into an array of conditional values to again store variability as local as possible to preserve sharing. To support arrays of different length though and fulfill our invariants, we support also variations of arrays. That is, an array of objects (`Object[]`) would be represented as a conditional array of conditional objects (`V<V<Object>[]>`). Type erasure in Java complicates the implementation, but this can be solved by inserting additional dynamic type checks.

We arrived at this design after considering several tradeoffs: Our representation can store variability more locally, avoiding that a single variation in an entry requires to copy the entire array; also load and store operations are simple and fast. Overheads are only encountered for arrays with different length in different configurations, which is less common than variability in values in our experience. An alternative design could loosen our invariants for arrays and create a single maximum-length array of conditional values (based on the length of the configuration with the longest array; `V<Object>[]`) and a shadow variable and extra instructions for bookkeeping and length checking, but we only expect marginal performance benefits from this more complicated design.

## 3.4 Control Transfer

After describing how to transform bytecode instructions within a given variability context, we now focus on how to transform control-flow related constructions that may change variability

contexts by splitting or 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 and track and change variability contexts. As introduced in Section 2.2, 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, as discussed, for example, for store instructions in Section 3.3.1. The challenge is now to propagate and change variability context to achieve a shared execution for all configurations with maximal sharing.

In this section, we explain how we structure the program in blocks with the same variability context, and how we transfer control and contexts among these blocks. Subsequently, we then discuss two important properties of our design: (1) that variational execution preserves behavior of the original program (*Correct Execution Property*) and (2) that control transfer among blocks is efficient (*Optimal Sharing Property*). Finally, we present some technical challenges and their solutions regarding stack values during control transfer.

### 3.4.1 VBlock

We group all instructions that are statically guaranteed to always share the same variability context at runtime in a *VBlock*. VBlocks are separated by *conditional* jumps, that is, jumps that may depend on conditional values, in which case we may "split" the execution. After executing multiple VBlocks we may "join" the execution in another VBlock with a broader variability context (such joining is rare in symbolic execution approaches). For example, String replacement of a smiley image (Line 9) in Listing 1 has a more restricted context than the `getHTMLHeader` call (Line 6) because Line 9 is only executed when `SMILEY` is `true`, whereas the later `getHTMLFooter` call is again shared among all configurations.

VBlocks are similar to basic blocks in traditional program analyses. However, unlike basic blocks, which group individual instructions together because they are always executed in sequence, VBlocks group basic blocks together because they always share the same variability context. Thus, there can be jumps inside a VBlock as long as they do not depend on conditional values and thus share the same variability context.

Bytecode instructions can be partitioned into VBlocks by merging basic blocks in a control-flow graph iteratively until a fixpoint is reached. A block $B_1$ can be merged with a successor $B_2$ if the jump between $B_1$ and $B_2$ is not conditional (e.g., `goto` or *if* statement with non-conditional expression)[1] and all predecessors of $B_2$ are in the same VBlock. The latter condition is needed to recognize potential join points, when a block can be reached from two different VBlocks. Hence, a VBlock can be terminated by either a conditional jump or an unconditional jump. A VBlock can end with an unconditional jump if, for example, while merging basic blocks to form VBlocks, basic block `A` has an unconditional jump to basic block `C`, while basic block `B` has a conditional jump to `C`. We cannot merge `A` and `C` into one VBlock because of the conditional jump from

[1] **Invariant 2** implies that all values evaluated in an `if` statement are conditional, however, as we will discuss later in Section 3.5, we can optimize the transformation to statically recognize values that will not depend on configuration options, including, in the simplest case, constants.

B. Thus, A and C have to be separated into two different VBlocks with an unconditional jump between them.

### 3.4.2 Execution Strategy

This subsection presents how VBlocks are used. We first outline the goals of using VBlocks to achieve splitting and joining execution. Then, we present a solution that achieves our goals and provide an example.

**Goals.** Whereas variational-execution approaches that modify interpreters (such as VarexJ) can track multiple instruction pointers and their variability contexts, we need to cope with the fact that the instruction pointer of an unmodified JVM can only point to a single location at a time. So instead of changing the control transfer mechanism of the JVM, we use VBlocks to organize and create the execution order we want. At a high level, we pursue the following:

- Both branches of a conditional jump can be executed under corresponding restricted contexts (we call them "subcontexts"). That is, we are able to split execution.
- The code after both branches of a conditional jump should be executed only once for mutually exclusive contexts. That is, we should join execution as early as possible.

**Context Propagation.** Using VBlocks, we modify control flow decisions and manipulate variability contexts to achieve splitting and joining. The key idea of our design is to associate each VBlock with a variability context (a fresh local variable). We dynamically update variability contexts along execution to keep track of which VBlock(s) can be executed next and under which context. At any point in a method's execution, all VBlocks with a *satisfiable* variability context (i.e., the proposition formula is satisfiable) can be executed. The order in which multiple VBlocks with satisfiable contexts are executed does not matter for correctness, but does matter for performance, as we will show in Section 3.4.3.

At a jump between VBlocks, we transfer the current block's variability context to the target block's context. If the jump is conditional, we split the current variability context and transfer the two mutually exclusive contexts to the two successor VBlocks of the jump. The split is determined by the partial configuration space in which the *if* statement's expression evaluates to true.

To describe the control transfer more precisely, let us denote the sequence of VBlocks as $b_0, b_1, \ldots, b_n (n \geq 0)$, where $b_0$ represents the entry node in the control flow graph and $b_n$ represents the exit node. Let us denote the variability context of a VBlock $b_i$ as $\phi(b_i)$ (stored in a fresh local variable for each VBlock).

- At the beginning of a method execution, we initialize $\phi(b_0)$ with the method context, and $\phi(b_i) = \mathit{False}$ for all other VBlocks to indicate that only the initial VBlock of the method can be executed.
- After executing a VBlock $b_i$, we remember its variability context $\Phi = \phi(b_i)$ and then set that variability to *False*, indicating that this block should not be immediately executed again. We subsequently propagate its prior variability context $\Phi$ as follows:

1. If the execution of VBlock $b_i$ ends with an unconditional jump (e.g., `goto` instruction) to another VBlock $b_j$, the context of $b_j$ is updated as a disjunction between the current context of $b_j$ and $b_i$'s prior context $\Phi$. A disjunction is required because the target block may already have been executable under a different context, which we now broaden to join executions.

$$\phi'(b_j) = \phi(b_j) \vee \Phi \tag{3.1}$$

2. If the execution of VBlock $b_i$ ends with a conditional jump with two possible target VBlocks $b_j$ and $b_k$,[2] we split the execution based on the condition of the jump (usually the top value on the stack representing result of evaluating an *if* statement's expression). Let us denote the variability context in which the jump condition indicates a jump to $b_j$ as $X$. For example, the condition of the first *if* statement in our WordPress example is $\langle SMILEY, 1, 0\rangle$, which indicates the *then* branch should be taken under context $X = SMILEY$. We update the variability contexts of $b_j$ and $b_k$ as follows, again considering potential joins:

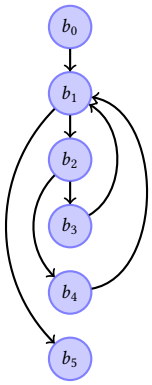
$$\phi'(b_j) = \phi(b_j) \vee (X \wedge \Phi) \qquad \phi'(b_k) = \phi(b_k) \vee (\neg X \wedge \Phi) \tag{3.2}$$

- After propagating the variability context, the control transfer (i.e., the actual instruction pointer in the JVM) does not actually follow the jump.

**Execution Order.** The actual execution order though (in terms of moving the instruction pointer) is independent from the transfer of variability contexts. We start execution at the beginning of the method with $b_0$. At the end of a VBlock $b_i$, we jump to the next VBlock $b_{i+1}$ by default, even if the block ended with a different jump. If that VBlock's variability context is unsatisfiable, we proceed to the next VBlock, and so forth. We only jump back to a VBlock with a lower index (using a plain `goto` instruction) when we update the variability context of an earlier block to be satisfiable as part of the described context transfer. This way, the instruction pointer is always at an unsatisfiable block (to be skipped) or at the satisfiable block with the lowest index. This strategy ensures that later VBlocks are always executed with joined variability contexts from earlier VBlocks and that VBlock $b_n$ is executed last with the full method context. For that reason, the indexing order of VBlocks matters. Figure 3.1 illustrates the idea of jumping among VBlocks with a concrete example.

**Ordering VBlock Execution.** Given that we always execute the first VBlock with a satisfiable variability context and always join at later VBlocks, we can execute the same method in different ways by changing the way we order the VBlocks. We can reorder VBlocks in different orders as long as the first and last VBlock remain constant (the last block ending with a return statement must be executed last) and always achieve equivalent (i.e., correct) results, as we will show in Section 3.4.3. However, as the block order determines the join points, different orders may be more or less effective at joining early and sharing subsequent computations.

[2]We transform switch statements into an equivalent series of *if-else* statements to simplify our design of control transfer.




| | | | |
|---|---|---|---|
| $b_0$ | `int i = 0;` | Update: $\phi(b_1)$ | Goto $b_1$ |
| | | Skip to $b_2$ unless $SAT(\phi(b_1))$ | |
| $b_1$ | `while (i < b) {` | Update: $\phi(b_2), \phi(b_5)$ | Goto $b_2$ |
| | | Skip to $b_3$ unless $SAT(\phi(b_2))$ | |
| $b_2$ | `i++`<br>`if (i != a)` | Update: $\phi(b_3), \phi(b_4)$ | Goto $b_3$ |
| | | Skip to $b_4$ unless $SAT(\phi(b_3))$ | |
| $b_3$ | `continue;` | Update: $\phi(b_1)$ | Goto $b_1$ |
| | | Skip to $b_5$ unless $SAT(\phi(b_4))$ | |
| $b_4$ | `...` | | |
| | `}` | Update: $\phi(b_1)$ | Goto $b_1$ |
| $b_5$ | `return;` | | |

Figure 3.1: An example illustrating control-flow encoding through updates of variability contexts and jumps between blocks.

To maximize sharing during the execution (i.e., prefer executing a block once under a broader variability context rather than multiple times under narrow contexts), we order VBlocks based on the *strict transitive predecessor* relation in the control-flow graph. A VBlock $b_i$ is a strict transitive predecessor of $b_j$ if there is a path from $b_i$ to $b_j$ in control-flow graph, but not from $b_j$ to $b_i$ (i.e., not in a loop). *For any pair of VBlocks, if one VBlock is a strict transitive predecessor of the other, the transitive predecessor shall have the lower VBlock index to be executed first.* For other pairs, we preserve the original lexical order produced by the compiler as a default.

In the next subsection, we will show that the above partial order is sufficient to statically guarantee optimal sharing on a subset of control-flow graphs, regardless of the original lexical order of the bytecode, but that optimality cannot be statically guaranteed for all control-flow graphs. We will also experimentally show in Section 3.6 that this order is nearly always optimal for the remaining control-flow graphs.

**Example.** Let us exemplify our solution by stepping through the `getWeather` method in Listing 2 (Figure 2.1 on page 10). There are four VBlocks: code before the `if` statement ($b_0$, Line 57-59), then branch ($b_1$, Line 60-64), else branch ($b_2$, Line 65-69) and return block ($b_3$, Line 70). These blocks are already indexed according to the strict transitive predecessor relation: $b_0$ is executed first, $b_1$ and $b_2$ are executed before $b_3$; the order between $b_1$ and $b_2$ is merely derived from the lexical order and could be switched. Initially, $\phi(b_0) = MCtx$ (method context) and $\phi(b_1) = \phi(b_2) = \phi(b_3) = False$. After executing $b_0$ at Line 59, $\phi(b_1)$ and $\phi(b_2)$ are updated to $\phi(b_1) = False \vee (FAHRENHEIT \wedge MCtx)$ and $\phi(b_2) = False \vee (\neg FAHRENHEIT \wedge MCtx)$, thus *splitting* the execution. Note that this update of contexts is not shown in Listing 2 because we transform the control flow in bytecode differently from how we show for Java. At this point, both $\phi(b_1)$ and $\phi(b_2)$ are satisfiable and execution continues with the next VBlock $b_1$. After executing $b_1$ at Line 64, $\phi(b_3)$ is updated to $\phi(b_3) = False \vee (FAHRENHEIT \wedge MCtx)$ because $b_3$ is the sole successor of $b_1$ in the control flow graph. We execute the next satisfiable block, which is $b_2$, after which $\phi(b_3)$ is updated to $\phi(b_3) = (\neg FAHRENHEIT \wedge MCtx) \vee (FAHRENHEIT \wedge MCtx) = MCtx$; thus, $b_3$ at Line 70 is executed last under the *joined* context $MCtx$.

### 3.4.3 Properties

We have presented how we choose VBlocks for execution. While splitting executions, we need to ensure that the execution order is correct. By always executing the satisfiable VBlock with the lowest index first and ordering VBlocks deliberately, we make sure that the joining happens as early as possible. This section formalizes these properties.

#### Correctness

The following property ensures that our variational execution is correct, in a sense that it preserves the semantics of the original program.

Property (Correct Execution Property). *At any point of execution, if there are multiple VBlocks with satisfiable contexts, the order in which they are executed does not affect correctness of execution.*

To prove this, we first introduce a useful lemma:

Lemma (Disjoint Context Lemma). *At any point of variational execution, the context of two different VBlocks are mutually exclusive. That is,* $\phi(b_i) \wedge \phi(b_j) = False$ *for any* $i \neq j$.

Mutual exclusion is guaranteed by the way we propagate contexts in Equations 3.1 and 3.2. A proof by induction can be found in Section 3.8. With this lemma, we prove our Correct Execution Property as follows:

*Proof.* Ensuring that VBlocks have mutually exclusive variability contexts guarantees that each VBlock operates on mutually exclusive runtime states. As we have discussed in Section 3.3, states (e.g., local variables, fields) are stored separately for different contexts using conditional values. Our variability contexts further ensure that all state changes only update values in the (disjoint) contexts referred to by the current variability context. Thus, execution order among satisfiable VBlocks does not affect correctness of overall variational execution. □

#### Optimal sharing

The main utility of variational execution is its ability to share common computations; our execution scheme pursues to perform executions with the broadest variability context possible. While we cannot share repeated executions under the same context, we can avoid executing the same VBlock under mutually exclusive contexts and rather execute it once, shared, under a broader context. In a nutshell, what we want to achieve is to execute every VBlock as few times as possible by sharing the execution of VBlocks in different contexts. This sharing is crucial for the overall performance of variational execution, otherwise it may degrade to executing each variation in a brute-force way or sharing only common prefixes of traces, conceptually equivalent to joining only after the very last instruction.

To formalize *optimal sharing*, we define a variational trace as a chronological sequence of VBlocks executed during variational execution. We denote a variational trace as a sequence of executed VBlocks with corresponding variability context, e.g., $t_v = [b_0^{True}, b_1^{\alpha}, b_2^{\neg\alpha}, b_3^{True}]$. Conceptually, a variational trace corresponds to a separate concrete trace for each configuration, in our example $t_\alpha = [b_0, b_1, b_3]$ and $t_{\neg\alpha}=[b_0, b_2, b_3]$. Another variational execution trace that

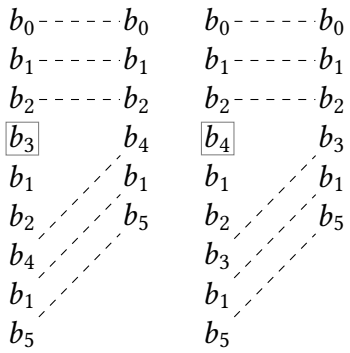



Figure 3.2: An example where static order between VBlocks cannot not achieve optimal sharing. The control-flow graph is shown in Figure 3.1

represents the same concrete traces could be $t'_v = [b_0^T, b_1^\alpha, b_3^\alpha, b_2^{\neg\alpha}, b_3^{\neg\alpha}]$. It is likely that $t_v$ is more efficient than $t'_v$ because $b_3$ is executed twice in $t'_v$.

A variational trace can be seen as the result of aligning multiple concrete traces. Different aligning schemes produce different variational traces (e.g., $t_v$ and $t'_v$). Given a set of concrete traces, we can use sequence alignment algorithms (e.g., Needleman-Wunsch algorithm [115]) to obtain a globally optimal solution of merging concrete traces. For example, two optimal matchings of $t_\alpha$ and $t_{\neg\alpha}$ are $t_o = [b_0, b_1, b_2, b_3]$ and $[b_0, b_2, b_1, b_3]$. We use $length(t)$ to denote the number of elements in a trace. For example, $length(t_v) = 4$, and $length(t'_v) = 5$.

Definition (Optimal Sharing). *Given a variational trace $t_v$ and its corresponding set of concrete traces $t_1, t_2, \ldots, t_m$, we say $t_v$ has optimal sharing if and only if $length(t_v) = length(t_o)$, where $t_o$ is the optimal matching of $t_1, t_2, \ldots, t_m$.*

It would be ideal if optimal sharing could be achieved for all possible programs in the wild, but there is no join strategy that could *statically* order blocks to guarantee optimal sharing for all executions of all programs. Figure 3.2 illustrates an example: In order to achieve optimal sharing (with optimal defined as the optimal trace alignment in the figure), there is both a case where $b_3$ needs to be executed before $b_4$ and a case where $b_4$ needs to be executed before $b_3$ after a control-flow decision at $b_2$ (critical nodes highlighted in the trace). That is, we cannot *statically* decide an ordering between $b_3$ and $b_4$, and even an optimal decision at runtime would have to depend on knowing the *future* execution trace. We could apply some greedy strategies to approximate optimality, but that the required runtime monitoring is unlikely to justify the performance benefits of additional sharing execution.

Fortunately, we can prove optimal sharing for static VBlock ordering for many shapes of control-flow graphs and will show in our empirical evaluation that the remaining ones (often with nontrivial interleaving of looping and branching instructions) are often optimal for actual executions.

Property (Optimal Sharing Property). *Given a control flow graph where each node represents a VBlock, our variational execution based on the strict transitive predecessor relation on this graph has optimal sharing if it is acyclic or only contains simple loops. A loop is a simple loop if it satisfies the following three criteria: (1) has only one loop header; (2) has only one exiting node; (3) has no conditional jumps among nodes in the loop.*

The proof can be found in Section 3.8. Intuitively, we prove by case analysis that our

variational trace has the same length as the optimal alignment of corresponding concrete traces in every possible case. Since we only consider simple control-flow graphs, the length of our variational trace and the length of the optimal alignment trace can be determined from the structure of the control-flow graph.

### 3.4.4 Values on the Stack between VBlocks

In Java, blocks can leave values on the operand stack to be consumed by subsequent blocks. Since, in variational execution, there might be multiple successor blocks that will be executed, and successor blocks may not be executed immediately after their predecessor, sharing values on the stack becomes tricky. Since the operand stack in a commodity JVM is not variational itself, we cannot pop the same value from the stack under different variability contexts as possible when modifying the interpreter itself (e.g., done in VarexJ [110]).

Figure 3.3 shows a concrete example in which VBlock $b_0$ leaves some value on the operand stack after execution, that both blocks $b_1$ and $b_2$ try to read. Conversely, those two blocks each leaves a (different) value on the stack that $b_3$ attempts to consume.

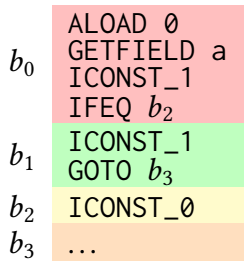



Figure 3.3: A snippet of bytecode showing the scenario where VBlocks could leave some values on the operand stack after execution.

To make the transition between VBlocks possible in all cases, we need one more invariant in addition to those introduced in Section 3.3:

**Invariant 4** A VBlock does not leave any values on the operand stack at the VBlock boundary.

To meet this invariant, we store all remaining values on the operand stack (if they exist) to local variables, at the end of each VBlock. Then at the beginning of each VBlock, we check if the current VBlock expects some values from the operand stack, and load those values from corresponding local variables if so. Since we support loading and storing under different variability contexts, as discussed above, this solution generalizes to all control-flow graphs.

## 3.5 Implementations, Optimizations, Limitations

We implemented a bytecode transformation tool for the ideas discussed in Section 3.3 and Section 3.4, and we call it VarexC. We use the ASM[3] library to implement our data flow analysis and transformation of bytecode. Transformations happen at class loading time via our own

[3] http://asm.ow2.org/

class loader that transforms classes before they are actually loaded. We also save the previously transformed classes and reuse them if there are no changes. To ensure correct implementation of variational execution, we apply differential testing to compare execution results and execution traces in variational execution against brute-force concrete executions for our subject systems [65]. Our implementation is available on GitHub.[4] We implement transformations for all bytecode instructions and provide a mechanism for model classes, as discussed in Section 3.3.2. In addition to implementing the full transformation described previously, we explored two optimizations and briefly outline them. Finally, we discuss the current limitations of our tool.

### 3.5.1 Optimization: Deciding What to Transform

Not all bytecode instructions in a program may depend on configuration options. If we can statically guarantee that parts of the program never depend on conditional values or conditional jumps, we can reduce our transformation to relevant parts; this reduces the overhead of computing with conditional values and conditional jumps where not needed. Guaranteed non-conditional computations happen often in the beginning of methods and typically involve initialization sequences or constants, such as in logging statements "`System.out.println("done");`".

We designed a simple data-flow analysis to decide which instructions need to be transformed. Along the lines of a standard taint analysis, we mark all local variables and values on the operand stack as conditional or unconditional with a 'lift' bit, marking them as conditional when instructions based on other conditional values write to them.

So far, we implemented an intra-procedural analysis that assumes all fields (including fields representing configuration options, as in our WordPress example) and method parameters and method results are conditional. As such, all stack values produced by field reads, loads of method parameters, and results from method invocations are marked with the lift bit. We then propagate the lift bit to all values resulting from computations in which operands had the lift bit and to local variables when such values are stored. Based on the lift bit, we decide which control-flow decisions are conditional (i.e., potentially depend on conditional values) and compute VBlocks correspondingly. Finally, we determine with a simple control-flow analysis, which VBlocks are guaranteed to be executed with the method's variability context (in a nutshell, all VBlocks that dominate the method exit) and mark all variables stored in other VBlocks as conditional, as they may be stored only in restricted contexts. As is common for data-flow analyses, we repeat these computations until a fixpoint is reached.

Based on our analysis, we transform bytecode based on a potentially smaller number of VBlocks (because some jumps are statically guaranteed to be non-conditional when their expression does not have the lift bit). We also transform only variables and instructions with the lift bit. We introduce additional instructions to translate concrete values into conditional values when values flow from unmodified into transformed code (i.e., just wrapping the concrete value in a `V` instance, boxing primitive types if necessary), such that our invariants still hold from the perspective of the transformed instructions.

Our current analysis is very conservative, because it assumes all fields and method signatures are conditional, thus most savings relate to constants and initialization sequences. Nonetheless, in

[4] https://github.com/chupanw/vbc

the programs of our evaluation (Section 3.6), we can statically decide to not lift up to 32.6 percent of all instructions, which however has only a marginal impact on performance. We hope that future work can push this analysis even further by performing an inter-procedural analysis to determine which methods and method arguments need to be transformed, potentially providing multiple transformed or partially transformed copies of the same method.

### 3.5.2 Optimization: Using Model Classes

As discussed in Section 3.3.2, we provide a mechanism for model classes with which we can implement custom implementations for classes where automated transformations are not possible (e.g., native methods, environment barrier) or inefficient. In fact, it is often possible to provide more efficient implementations of common data-structure implementations that are specifically designed for variability [112, 158]. Our model-class mechanism allows drop-in replacements for such classes.

#### Variational data structures

We implemented a small number of custom variational data structures for commonly used collections. For example, instead of an automated transformation of the `java.util.LinkedList` class, which would support conditional values and conditional successors of linked-list nodes, we use a custom implementation that internally stores a list of optional elements and provides corresponding accessor functions. Similarly, rather than automated transformation of `java.util.HashSet` objects, we can represent variational sets as a mapping from values to variability contexts that describe the configuration space in which the set contains that value. As explored by Walkingshaw et al. [158], such tailored representations are often (though not generally) much more efficient, especially when they hold many optional elements with different conditions [112, 158].

Depending on different computations in different programs, the effectiveness of model classes varies. For example, our model `LinkedList` is optimized for iterating elements, so programs that iterate lists of optional entries frequently gain more benefits from our model classes. Our evaluation shows different levels of improvement after the drop-in replacement of some model classes, with up to 6 times speedup for GPL.

#### Custom access patterns

While custom data structures can store conditional entries more efficiently, common accessor patterns to iterate over list entries can still be very inefficient. For example, getting first the first, then the second element of a list with optional entries $[1^{\alpha}, 2^{\beta}, 3^{\gamma}, 4, 5]$ would create large conditional values (e.g., $\langle \alpha, 1, \langle \beta, 2, \langle \gamma, 3, 4 \rangle \rangle \rangle$ for the first element).

Instead, we detect common access patterns and transform them more intelligently. Instead of iterating over all elements of the list one by one (where each element can be a conditional value), we iterate over all optional elements, where the element is a concrete value, but the iteration is executed under a restricted variability context based on that element's condition, which marks under what context this element exists in the list. We integrate the most recent detection and

rewrite of such access patterns of Lazarek [84]. Our current implementation detects loops that use `iterator` and automatically transform such loops to use our specialized list more efficiently.

In our evaluation, there were only few instances that benefited from this optimization, but if they did, the improvements were substantial. For example, in *CheckStyle* (see Section 3.6), the program iterates over a list of 135 optional checks. The basic transformation results in exponential behavior, that makes it infeasible to execute the code without manual rewrites of the CheckStyle implementation, whereas our optimization of access patterns allows to execute this code fragments efficiently. Overall, several researchers have explored variational data structures and access patterns recently [112, 158]. Model classes and additional rewrites during the transformation allow us to easily integrate such advances to improve performance of variational execution on real-world systems.

### 3.5.3 Limitations

Our current implementation of variational execution has some limitations, most of which are related to low-level details of the JVM or restrictions posed by the Java runtime (e.g., we cannot directly modify classes in the `java.lang` package for safety reasons). Most limitations are engineering challenges that can be overcome with additional implementations, typically in the form of model classes.

**Exception.** We distinguish two types of exceptions: non-variational exceptions and variational exceptions. *Non-variational exceptions* are thrown or propagated under the current method context. *Variational exceptions* are thrown or propagated under a smaller context than the current method context (i.e., only in some partial configurations). Semantically, non-variational exceptions represent cases where invoking a method under `ctx` would always result in the same exception under *all configurations* of `ctx`, whereas variational exceptions occur *only in some partial configurations*.

Non-variational exceptions are easy to support because the control flow is the same for all configurations. In fact, we only found non-variational exceptions when executing our subject systems in evaluation.

Variational exceptions are trickier to handle because method execution might be interrupted under some partial configurations. If the exception is not caught inside method invocation, returning from a method results into a normal return value in some configurations and an uncaught exception in other configurations. Although it is possible to support variational exceptions by delaying throwing them and wrapping them together with normal return values as a conditional value, the transformation would complicate the control flow of transformed bytecode in a nontrivial way, especially if exceptions are supposed to be caught inside the current method or some outer methods. Since variational exceptions are not that common in our experience, we adopt a less efficient but easier approach to support variational exceptions: The key idea is to throw an exception immediately when it occurs and continue the rest of the variational execution only under the variability context of the exception; then we restart variational execution under the remaining contexts that did not result in the previous exception, and keep repeating until all contexts have been explored. Re-executions might affect overall efficiencies of variational execution, but we only observed variational exceptions in our own

artificial examples. In cases where not all exceptions are important to capture, we can simply ignore the paths that throw exceptions and continue variational execution under restricted contexts that represent exception-free paths. In this chapter and Chapter 4, we adopt the re-execution approach to exception handling in order to accurately handle all exceptions, but in Chapter 5, we avoid re-execution and focus on paths that execute normally without exceptions, which we will discuss in greater length in Section 5.5.

**Model Classes.** We only implemented a handful of model classes (9 classes and in total 1030 lines of Java code) to tackle the environment barrier required by our subject programs. We consider all classes that have native methods and classes that are closely related to internals of the JVM to be behind the environment barrier and use the strategies discusses above, including repeated invocations and model classes.

Currently we support a large set of Java programs, but we may need to provide more model classes if another program uses certain advanced language features. We adopt an incremental approach, in which we carefully monitor the need for model classes at runtime (i.e., when conditional values are passed across environment barriers). When implementing model classes, our main focus is to support conditional values. Symbolic execution and model checking face a similar challenge, but we argue that the implementation effort is lower for variational execution, because we compute with concrete values and can therefore delegate to existing implementations rather than reimplement abstractions of those operations.

**Reflection.** Reflection is relatively simple to handle due to the dynamic nature of our approach. Since reflection cannot modify bytecode (i.e., we cannot introduce conditional instructions at runtime), it does not affect our bytecode transformation of classes. We intercept reflection calls and replace them with our special call stubs, where we wrap arguments into conditional values, append variability context to the argument list, and invoke the transformed method, just as we would transform bytecode statically. We have implemented partial support for reflection as needed by our subject systems incrementally.

**Synchronization.** Two instructions (`monitorenter` and `monitorexit`) are used to synchronize concurrent operations. We currently keep them as is, which implies that we lock sections for all configurations, not just in the current variability context; this may lead to over-synchronization and potential liveness issues.

**Array.** As we will see in Section 3.6, array operations are generally expensive in their current form. Especially when crossing the environment barrier, we may need to translate conditional arrays into plain concrete arrays, which can be relatively expensive if the arrays are large. A more efficient implementation of variational arrays would be future work.

**Comparing to VarexJ.** Our approach comes with its own limitations, but most of them can be improved with additional engineering effort. We sidestep most bottlenecks of the state-of-the-art approach (VarexJ) by transforming bytecode instead of modifying the underlying JVM. Although the limitations of VarexJ can potentially also be removed by more engineering effort, we argue

that the effort in VarexJ is much higher because of those additional complexities from the JVM itself. As an example, native methods are notoriously difficult to support in Java PathFinder (JPF), the underlying JVM of VarexJ, largely because JPF has its own memory model for objects, which cannot be passed to native methods directly and therefore require additional conversion. In contrast, native methods can be supported in our approach by providing simple model classes to handle conditional values so that concrete values can be passed to native methods.

## 3.6 Empirical Evaluation

In an empirical evaluation, we now execute a number of configurable systems to assess performance (time and memory consumption) and effectiveness of sharing. Specifically, we compare our implementation against repeatedly executing the unmodified code in all configurations (brute-force execution)[5] and against VarexJ [110], a state-of-the-art variational execution engine for Java, which executes bytecode with a modified virtual machine based on Java Pathfinder. While performance measures implicitly indicate the benefits of sharing, we additionally empirically assess how often our VBlock ordering results in optimal sharing at runtime, especially for methods for which we cannot guarantee optimal sharing statically.

### 3.6.1 Experimental Setup

**Benchmarks.** Table 3.1 shows the benchmark programs used in this study. For comparability, we use the same set of benchmark programs from VarexJ [110], which includes programs from various domains: Jetty 7 is a HTTP server; Checkstyle is a static coding style checker for Java programs; Prevayler is an in-memory database system; QuEval is an academic evaluation framework for database index structures; Elevator, GPL and E-Mail are commonly used benchmarks from the software product-line community that are designed to have many variations. These programs have 6 to 141 options, each of which is a `boolean` controlling inclusion or exclusion of a feature. Feature combinations are usually restricted by a feature model [142]. The goal of analyzing these programs is to estimate the effort of exploring a big configuration space, which can be useful for testing, static analysis, and so forth. We execute each program with a representative input, which in each system covers all configuration options and significant parts of implementation. For example, we feed Checkstyle with 4 Java source code files and use 135 different checkers to check coding style.

**Implementation and Hardware.** To compare with our tool VarexC, we use the latest VarexJ code base as of 12/12/2017, which includes the most recent optimizations, added after the last publication. Both VarexC and VarexJ are executed with Java HotSpot™ 64-bit Server VM (v1.8.0_161). We use a laptop with 2.30GHz Intel Core i7 CPU and 16GB system memory. All results are measured when the machine is idle and unloaded.

[5] For benchmark programs that have more than 20 configuration options, we randomly select 1 million valid configurations for measuring.

**Performance Measurement.** We measure performance in three different settings:

- First, we measure the performance of executing the unmodified program in every single configuration separately on a commodity JVM. Since the execution time may differ significantly between configurations, we report both the average execution time (reported as $\mu$JVM) and the execution time of the slowest configuration (reported as max JVM).
- Second, we measure the time it takes VarexJ, the state of the art variational interpreter built on top of Java Pathfinder, to execute the program across all configurations (reported as *VarexJ*).
- Finally, we measure how long it takes to execute the program across all configurations by executing the modified bytecode with a commodity JVM (reported as *VarexC*).

Ideally, the performance of variational execution (*VarexJ* and *VarexC*) would be between the execution time of the slowest configuration (max JVM) and the combined execution time of all configurations ($\mu$JVM $\cdot$ number of configurations): Variational execution needs to at least execute all instructions of the slowest configuration, but it can usually share effort among multiple configurations.

In all three cases, we measure steady-state performance for each benchmark, based on repeated executions [39]. Steady-state measurement excludes JVM startup time, which typically dominates by JIT compilation and class loading. We do not compare startup performance because VarexJ is implemented as a Java interpreter itself—in addition to loading classes of benchmark programs, VarexJ needs to load a lot of necessary classes for the meta-circular interpreter to work, which would bias our results against VarexJ. For VarexC, we exclude the bytecode transformation time from measurement because transformation happens once for each program, similar to compiling source code. We only measure VarexC with *all optimizations* (see Section 3.5) for brevity. Following the suggestion from Georges et al. [39], we measure steady-state performance in the following steps:

1. Start a JVM invocation $i$ and iterate the benchmark until a steady-state is reached, i.e., once the coefficient of variation (CoV) of 10 consecutive iterations falls below a predefined threshold, which is 0.02 in our case.
2. For the JVM invocation $i$, compute the mean execution time of those 10 steady iterations, and denote it as $\bar{x}_i$.
3. Repeat Step (1) and (2) for 10 times and compute the overall mean $\bar{x} = \frac{\sum_{i=1}^{10} \bar{x}_i}{10}$. Finally, we report $\bar{x}$ as the measurement result.

In the above measurement, Step (1) and (2) are designed to warm up the JVM, excluding factors like class loading and JIT compilation. These factors are less interesting to our evaluation because our main goal is to measure performance of variational execution. The coefficient of variation threshold is useful for controlling the effect of garbage collection. Step (3) is designed to minimize non-determinism of JIT compilation across JVM invocations, because JVM uses timer-based sampling to drive JIT optimization (e.g., which methods to optimize, at what level). Other main sources of non-determinism include thread scheduling and garbage collection. Thread scheduling is less of a concern for us because all programs except Jetty are single-threaded. Regarding Jetty, we configure Jetty to run a small server that has minimal thread scheduling. Georges et al. [39] recommends reporting a confidence interval instead of the mean alone. However, as we will show, the performance difference between our approach and VarexJ is so

large that reporting confidence intervals is unnecessary. The difference is so obvious and the variation so small in comparison that statistical tests are not needed. Due to this large effect size, we omit confidence intervals for brevity.

**Memory usage.** To measure memory usage, we calculate the used heap space by calling APIs of `java.lang.Runtime` at *every method entry*, and then record the *maximum* heap space used throughout the entire JVM invocation. Even with this frequent sampling, we cannot guarantee accurate measurement of memory usage, largely because of the non-deterministic garbage collection and bulk memory allocation. Thus, the memory measurement is only useful for coarse-grained comparison. For VarexC and VarexJ, we perform each single measurement on a given subject program by executing it *once*. As a comparison goal, we also measure the memory usage of executing one representative configuration on a commodity JVM (reported as *JVM*). The representative configuration is chosen as a valid configuration with the most features enabled. Since VarexC and VarexJ explore the entire configuration space, their memory consumption is strictly larger than execution of a single configuration. To reduce noise, we repeat each measurement 10 times and report the average.

**Sharing Efficiency.** As discussed in Section 3.4.3, our approach is able to give static guarantees of optimal sharing to methods that satisfy certain conditions. To assess sharing for other methods, we monitor the sharing in our benchmark executions. Specifically, we collect traces of which VBlocks are executed under which conditions and subsequently analyze whether those traces were optimal, with regard to sharing. For each variational trace, we expand it into a set of all distinct concrete traces that it represents, and then compute the alignment of these concrete traces. Since an optimal alignment of $n$ traces is NP-hard [160], we compute pairwise alignments between all distinct concrete executions using Needleman-Wunsch algorithm [115]. If the observed variational trace is longer than the longest pairwise alignment, we consider the sharing as not optimal. This pairwise approximation is conservative in that we may consider executions with optimal sharing as not optimal if the n-way alignment is longer than the longest pair-wise alignment; conversely, if the variational trace is not longer than the longest pair-wise alignment we can be sure that the sharing is optimal. Our pairwise alignment approach sidesteps the need of computing optimal alignment, but it still has scalability issues if there are too many pairs, which happen sometimes in our evaluation. For those cases, we conservatively mark them as suboptimal.

### 3.6.2 Execution Time

Table 3.1 summarizes the performance results, showing that VarexC outperforms VarexJ by a factor between 2 to 46. Variational execution is obviously significantly slower than executing a single configuration (between 4 and 3200 times slower), but as configuration spaces grow exponentially, this slowdown is often practical to cover the entire space.

**VarexC vs. VarexJ.** Comparing VarexC and VarexJ, we can see that VarexC outperforms VarexJ in all cases, with a speedup of 2 to 46. To better understand the speedup, it is useful to divide our subject programs into two groups and discuss them separately.

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 | $\mu$JVM (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 |

QuEval, GPL, Elevator, E-mail are academic examples that only need basic language features, such as arithmetic computation and array operations. Thus, a comparison between VarexC and VarexJ on these programs reveals the performance gap between bytecode transformation and interpreter instrumentation. As we can see, we are up to 10 times faster than VarexJ, due to lower interpreter overhead and JVM optimizations. QuEval is dominated primarily by heavy computations with arrays with only moderate sharing that are expensive in both VarexC and VarexJ. In a micro-benchmark, we confirmed that sorting on an array of 1000 variational elements with VarexC is roughly 2 times faster than VarexJ, which likely explains the low performance difference for this program.

Jetty, Prevayler, Checkstyle are medium-sized real-world programs that are widely used in practice. These programs use various more advanced JVM features, including dynamic class loading (CheckStyle), network access (Jetty) and file access (Prevayler). Since VarexJ is built upon a research JVM, it inherits limitations from its underlying JVM in this regard, whereas code transformed with VarexC remains portable across JVMs.

**VarexC vs. Individual Executions.** 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 and worst-case configurations.

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.

QuEval, GPL, Elevator, Email represent extreme cases where variations are used heavily. As we can see from Table 3.1, up to 940 configurations are encoded in merely 3,109 lines of code for QuEval. When program variations (we called them features interchangeably) present compactly, the sharing of data and execution becomes less frequent, and thus explains why VarexC and VarexJ cannot outperform brute force because variational execution relies on sharing to be efficient. In fact, there is a loop in Checkstyle that causes state space explosion for VarexJ because

Table 3.2: Memory usage comparison of JVM, VarexJ and VarexC.

| Subject | JVM (in MB) | VarexJ (in MB) | VarexC (in MB) | VarexJ/ VarexC | VarexJ/ JVM | VarexC/ JVM |
|---|---|---|---|---|---|---|
| Jetty | 268 | 2,739 | 648 | 4.2 | 10.2 | 2.4 |
| Checkstyle | 504 | 1,106 | 835 | 1.3 | 2.2 | 1.7 |
| Prevayler | 65 | 1,378 | 288 | 4.8 | 21.1 | 4.4 |
| QuEval | 59 | 301 | 282 | 1.1 | 5.1 | 4.8 |
| GPL | 141 | 342 | 151 | 2.3 | 2.4 | 1.1 |
| Elevator | 58 | 92 | 67 | 1.4 | 1.6 | 1.2 |
| E-Mail | 59 | 67 | 67 | 1.0 | 1.1 | 1.1 |

of looping a list that has $2^{135}$ variants. VarexC uses a model class to handle this loop gracefully, as discussed in Section 3.5. However, unlike these extreme cases, programs in practice often adopt separation of concerns and thus features do not interact very heavily all the time [110].

Jetty, Prevayler, Checkstyle implement configuration options such that they are often orthogonal to each other or have relatively local effects, which facilitates sharing better. As we can see in Table 3.1, by exploiting sharing, the performance of VarexC for exploring the entire configuration space is even relatively close to executing only the slowest configuration, with a slowdown as small as a factor of 4.

**Verdict.** We argue that the runtime overhead of VarexC is reasonable except for one case (QuEval) where expensive array operations with little sharing dominate the performance. 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 [110, 134]. 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.

**Summary**

VarexC outperforms VarexJ with a speedup of 2 to 46 times. The performance gain comes from various factors, including further optimizations at low level and portability to mature JVM implementations, all of which benefit from our strategy of transforming bytecode instead of modifying a language interpreter. Moreover, VarexC is performant and efficient for practical use in analyzing the whole configuration space of programs.

### 3.6.3 Memory Usage

Table 3.2 summarizes the memory usage results, showing that VarexC is more memory efficient than VarexJ in all cases except for a tie in E-Mail. Conceptually, VarexC and VarexJ perform

Table 3.3: Sharing efficiency of VarexC. We analyze methods both statically and at runtime. At runtime, we distinguish between method executions that are statically guaranteed to be optimal, that are dynamically observed to be optimal, and that are dynamically observed to be not optimal.

* Our alignment analysis has scalability issues with some variational traces of Checkstyle, mainly because there are too many features (up to 130) in each single trace, resulting into too expensive pairwise alignment. For those variational traces, we conservatively report them as non-optimal.

| | Method analysis (static) | | Method execution (dynamic) | | |
|---|---|---|---|---|---|
| Subject | Guaranteed Optimal | No Guarantee | Guaranteed Optimal | Observed as Optimal | Observed as Non-Optimal |
| Jetty | 2,667 | 257 | 19,043 | 3,734 | 0 |
| Checkstyle | 2,878 | 281 | 1,992,879 | 268,689 | *230 |
| Prevayler | 722 | 108 | 58,274 | 5,036 | 0 |
| QuEval | 458 | 103 | 57,383 | 8,920 | 267 |
| GPL | 244 | 44 | 34,641 | 3,476 | 0 |
| Elevator | 119 | 13 | 2,453 | 218 | 1 |
| E-Mail | 314 | 40 | 2,264 | 120 | 0 |
| Total | 7,402 | 846 | 2,166,937 | 290,193 | 498 |

a similar computation, so the extra memory consumed by VarexJ could result from two main aspects: less efficient sharing in data and the overhead of the underlying meta-circular interpreter. As the differences are fairly consistent across benchmarks, we attribute most efficiency gains to the interpreter's overhead rather than to differences in sharing. Both VarexC and VarexJ, as expected, consume more memory when compared to the execution of a single configuration, with the gaps noticeably smaller for VarexC. The memory overhead of VarexC largely comes from analyzing other configurations. We argue that the extra memory overhead shown in Table 3.2 is acceptable for modern machines.

**Summary**

VarexC is more memory efficient than VarexJ, due to more efficient sharing in data and less overhead from the implementation. Moreover, VarexC has the memory efficiency to analyze the entire configuration space in practice.

### 3.6.4 Sharing Efficiency

Table 3.3 shows how efficient our sharing of VBlocks is in practice. As we can see, we can make static guarantees for 89.7 percent of all the methods in our benchmark programs. When observing the executions, those methods with static guarantees account for 88.2 percent of the executed methods, and we observed that 99.8 percent for the remaining ones were optimal as well. The number of method executions that redundantly execute VBlocks with suboptimal sharing is minimal.

**Summary**

Sharing in VarexC is efficient, with static guarantees to 89.7% of all methods. For methods with no static guarantees, VarexC achieves runtime optimality for 99.8% of those method invocations.

## 3.7 Related Work

We implement variational execution by transforming bytecode.

**Variational execution.** Variational execution is a technique to execute a program for different values while sharing common computations as far as possible. It has similarities with model checking and symbolic execution, but performs concrete executions, where multiple concrete values are distinguished with conditions external to the program, and focuses on maximizing sharing during the execution by storing variations in data locally and by aggressively merging control-flow differences. 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 similar inputs, typically to observe the similarities and differences among executions, often with the focus on interactions among multiple differences.

- A common use case is testing configurable systems, in which a single test case should be executed over a large configuration space. For example, Nguyen et al. [117] used variational execution to render the content of WordPress while controlling how various plugins interact and affect the execution; Meinicke et al. [110] and Kim et al. [69] 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 [66, 69, 117]. Soares et al. [146] furthermore used differences among executions as clues to find suspicious feature interactions. Reisner et al. [134] 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 10KLOC programs) due to limited sharing abilities of symbolic execution [110].
- Austin and Flanagan [7] uses variational execution (under the name faceted execution) to track information flows in a program. In this context, the program is evaluated with sensitive and nonsensitive values at the same time, where the equivalent of options are decisions who is allowed to see which value. In contrast to prior multi-execution work which observes differences between two executions, Austin's analysis based on variational execution can track interactions among multiple decisions. This line of work has been extended with models for variational database storage [171]. There are also libraries to enable developers to directly write variational programs for this information-flow analysis, rather than relying on a variational execution engine [8, 140, 141].
- Variational execution can further be used to explain the differences in program executions among multiple inputs [109], in line with delta debugging [82, 149, 173].

- Variational execution is potentially useful for approaches that speculatively change source code or execution to evaluate the consequences. For example, mutation testing [57] and heuristics-based automatic program repair [90] typically try many small changes to the source code and re-execute the test suite for each change to evaluate test suite quality or find patches (Chapter 5). Zhang et al. [174] speculatively switches predicates in program and re-executes the program to detect execution omission errors. Brun et al. [18] proactively merges different versions and repeatedly executes the test suite to detect collaboration conflicts early. By encoding changes as variations, variational execution can explore the effects of changes efficiently and uncover interesting interactions of changes [165].
- Finally, variational execution can be used to speed up similar computations if there is sufficient sharing to offset the overhead. For example, Sumner et al. [148] shares similarities among executions of simulation workloads and computes with several values in parallel. Wang et al. [159] 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. [155] executes patched and unpatched programs together to share redundant computations when testing a patch. Variational execution has the potential to scale such use cases to exploring interactions among multiple changes.

Variational execution is fundamentally different from traditional approaches of multi-execution [27, 29, 50, 76, 147] and delta debugging [82, 149, 173] 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 execution exploits sharing and allows to observe differences among executions during the execution.

Ideas similar to variational execution can be found also in approaches for model checking and symbolic execution [25, 143, 157], 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. [110] has shown, current approaches are less effective at sharing than the aggressive sharing in variational execution.

**Implementing Variational Execution.** Existing variational execution approaches (and related approaches) are typically implemented by modifying the execution engine [7, 12, 66, 69, 105, 110, 143], typically research prototypes or metacircular interpreters that cause significant overhead and provide only limited support for all language features. Schmitz et al. [140, 141] provided a library for Haskell with which users can directly implement programs to use variational execution, similar to our example in Listing 2 of Figure 2.1 on page 10.

Instead, we pursue an approach in which we transparently modify Java bytecode to achieve variational execution on a commodity JVM. Our approach was inspired by Phosphor [13], a dynamic taint analysis for Java that tracks taints by instrumenting bytecode. In contrast to Phosphor, our modifications are significantly more extensive, as we need not only track additional data, but entirely change how computations and control flow happen in the program. CROCHET allows to explore different inputs to the same function by modifying bytecode to perform

checkpoints and rollbacks on the heap of a commodity JVM [14]. Comparing to CROCHET, our approach can achieve a more fine-grained sharing of executions while exploring different alternative values. The only other approach to execute programs variationally with commodity infrastructure is the implementation behind Jeeves [171], that uses metaprogramming to achieve similar changes for a small subset of Python. Their transformations are incomplete and not described beyond their implementation for a small example program.

**Quality assurance for configurable systems.** A main goal of variational execution is testing configurable systems. There are a wide range of approaches to analyze configurable systems with large configuration spaces, typically focused on reusing test cases across product variants, on sampling and on static analysis [36, 108, 119, 131, 152]. Sampling strategies analyze or execute a subset of configurations, but such analysis is neither exhaustive nor does it allow to easily compare executions [119]. For static analyses (including type checking, model checking, and data-flow analysis), researchers have explored many sharing strategies to encode variability locally (e.g., alternative types for expressions), to reason about large configuration spaces with propositional formulas, and to join computations early [92, 152]. In a sense, variational execution can be seen as a generalization of these sharing techniques for an interpreter [66]. Bodden et al. [16] and Dimovski et al. [31] describe how to lift existing static analyses by providing a variational framework on how to execute them.

## 3.8 Proofs

Lemma (Disjoint Context Lemma). *At any point of execution, the contexts of two different VBlocks are mutually exclusive. That is, $\phi(b_i) \wedge \phi(b_j) = False$ for any $i \neq j$.*

*Proof.* We prove by induction and case analysis on the jumping targets of a given VBlock. In the following, we use $b$ to denote a VBlock, $\phi(b)$ to denote the variability context of $b$, and $\phi'(b)$ to denote the new context after context propagation.

**Base Case.** At the beginning of execution, only the entry VBlock has a non-false context. Thus, $\phi(b_i) \wedge \phi(b_j) = False$ because at least $\phi(b_i)$ or $\phi(b_j)$ equals *False*.

**Induction Step.** Suppose before execution step $k$, $\phi(b_i) \wedge \phi(b_j) = False$, for any $i \neq j$. After execution of the next VBlock, say $b_l$, we need to update the context of $b_l$'s jumping targets.

- If $b_l$ has only one jumping target $b_m$, according to our context propagation, $\phi'(b_l) = False$, $\phi'(b_m) = \phi(b_l) \vee \phi(b_m)$. Obviously, $\phi'(b_l)$ is mutually exclusive to other VBlock context. For any VBlock context, say $\phi(b_o)$:

$$\begin{aligned} \phi'(b_m) \wedge \phi(b_o) &= (\phi(b_l) \vee \phi(b_m)) \wedge \phi(b_o) \\ &= (\phi(b_l) \wedge \phi(b_o)) \vee (\phi(b_m) \wedge \phi(b_o)) \end{aligned} \tag{3.3}$$

  According to our induction hypothesis, we have $\phi(b_l) \wedge \phi(b_o) = False$ and $\phi(b_m) \wedge \phi(b_o) = False$, thus induction hypothesis holds after execution of $b_l$.

- If $b_l$ has two jumping targets $b_m$ and $b_n$, splitting the execution on condition $X$, after executing $b_l$, we have $\phi'(b_l) = \mathit{False}$, $\phi'(b_m) = \phi(b_m) \vee (X \wedge \phi(b_l))$ and $\phi'(b_n) = \phi(b_n) \vee (\neg X \wedge \phi(b_l))$. For any VBlock context, say $\phi(b_o)$:

$$\begin{aligned} \phi'(b_m) \wedge \phi(b_o) &= (\phi(b_m) \vee (X \wedge \phi(b_l))) \wedge \phi(b_o) \\ &= (\phi(b_m) \wedge \phi(b_o)) \vee (X \wedge \phi(b_l) \wedge \phi(b_o)) \end{aligned} \tag{3.4}$$

  According to our induction hypothesis, we conclude that $\phi'(b_m) \wedge \phi(b_o) = \mathit{False}$. Similarly, we can conclude $\phi'(b_n) \wedge \phi(b_o) = \mathit{False}$. Moreover:

$$\begin{aligned} \phi'(b_m) \wedge \phi'(b_n) &= (\phi(b_m) \vee (X \wedge \phi(b_l))) \wedge (\phi(b_n) \vee (\neg X \wedge \phi(b_l)) \\ &= (\phi(b_m) \wedge \phi(b_n)) \\ &\quad \vee (\phi(b_m) \wedge \neg X \wedge \phi(b_l)) \\ &\quad \vee (\phi(b_n) \wedge X \wedge \phi(b_l)) \\ &\quad \vee (X \wedge \phi(b_l) \wedge \neg X \wedge \phi(b_l)) \end{aligned} \tag{3.5}$$

  Again, our induction hypothesis guarantees that $\phi'(b_m) \wedge \phi'(b_n) = \mathit{False}$. Thus, induction hypothesis holds after execution of $b_l$.

□

Property (Optimal Sharing Property). *Given a control flow graph where each node represents a VBlock, our variational execution on this graph has optimal sharing if it is acyclic or only contains simple loops. A loop is a simple loop if it satisfies the following three criteria: (1) has only one loop header; (2) has only one exiting node; (3) has no conditional jumps among nodes in the loop.*

As discussed in Section 3.4.2, the actual variational traces generated by our approach are influenced by the lexical order of VBlocks in the bytecode. To help us focus on the essential ideas of proving optimality on control-flow graphs, we introduce one precondition to the lexical order.

Precondition. *We assume that the strict transitive predecessor relation aligns with the lexical order of VBlocks in the bytecode. That is, for any pair of VBlocks $b_i$ and $b_j$, if $b_i$ is a strict transitive predecessor of $b_j$, $b_i$ precedes $b_j$ in the lexical order of bytecode.*

We also introduce two useful lemmas.

Lemma 1. *For any two concrete executions of the same simple loop expressed as traces of VBlocks, the shorter execution is a prefix of the longer execution.*

*Proof.* We prove by contradiction. Let us denote the shorter execution as $[x_1, x_2, \ldots, x_m]$, and the longer execution as $[y_1, y_2, \ldots, y_n]$, where each $x_i$ or $y_j$ represents a VBlock in the control-flow graph and $m \leq n$. Since a simple loop has only one loop header and one exiting node, $x_1$ must be the same as $y_1$, and $x_m$ must be the same as $y_n$.

For the shorter trace, let us assume it differs from the longer trace at the element $x_i$ (the $i-th$ element). Thus, $[x_1, x_2, \ldots, x_{i-1}]$ is the same as $[y_1, y_2, \ldots, y_{i-1}]$. Since $x_i$ is different from $y_i$, there must be a conditional jump at $x_{i-1}$ that jumps to either $x_i$ or $y_i$ in the control-flow graph.

This is contradicting the simple loop criterion that there are no conditional jumps among nodes in the loop.

□

LEMMA 2. *For any variational execution of a simple loop, the variational trace is a prefix of the longest concrete execution trace it represents.*

*Proof.* We prove by contradiction. Let us denote the variational execution as $[v_1, v_2, \ldots, v_m]$, and the longest concrete execution as $[x_1, x_2, \ldots, x_n]$, where each $v_i$ or $x_j$ represents a VBlock in the control-flow graph. Elements of a variational trace use superscripts to indicate variability contexts of execution, but they are less important in this proof so we omit them for brevity. Since a simple loop has only one loop header and one exiting node, $v_1$ must be the same as $x_1$, and $v_m$ must be the same as $x_n$.

Let us assume the variational trace differs from the longest trace at the element $v_i$ (the $i-th$ element). Thus, $[v_1, v_2, \ldots, v_{i-1}]$ is the same as $[x_1, x_2, \ldots, x_{i-1}]$. Since $v_i$ is different from $x_i$, there could be two causes. First, there is a conditional jump at $x_{i-1}$ that jumps to either $v_i$ or $x_i$ in the control-flow graph. Second, during variational execution of the loop, two different VBlocks have satisfiable contexts, which also requires at least a conditional jump among VBlocks in the loop because conditional jumps are the only places where we split variability contexts. Both of these cases contradict the simple loop criterion that there are no conditional jumps among nodes in the loop.

□

With the precondition and lemmas above, we will prove the original property below. Again, we prove that, given a control flow graph of VBlocks, our variational execution on this graph has optimal sharing if it is acyclic or only contains simple loops.

*Proof.* We prove by case analysis on acyclic control-flow graphs and control-flow graphs with simple loops, respectively.

**Acyclic.** For any acyclic control-flow graph, suppose our variational execution generates a trace $t_v$ with $n$ elements. Our static partial ordering between VBlocks ensures that these $n$ elements are different. Otherwise, suppose $b_i$ appears twice in $t_v$, there must be a transitive predecessor of $b_i$ between these two appearances of $b_i$ in $t_v$ because the control-flow graph is acyclic. However, this is impossible because $b_i$'s transitive predecessors can only precede $b_i$ in our variational traces, due to our static partial ordering.

As discussed in Section 3.4.3, $t_v$ represents a set of concrete execution traces under different restricted contexts. These concrete traces have the following two properties:

- There is no duplicated VBlock in each concrete trace, because the control-flow graph is acyclic.
- The $n$ different VBlocks in $t_v$ must appear in one or more of these concrete traces, because our variational execution only executes VBlocks with satisfiable contexts.

We denote the optimal sharing of these concrete traces as $t_o$. From these two properties, we know that $length(t_o) = n$ because each VBlock must occur at least once, and at most once if the traces are optimally aligned. So, the length of the optimal alignment must be $n$. Since $length(t_v)$ is also $n$, we achieve optimal sharing.

**Simple Loop.** For any control-flow graph with one or more simple loops, we denote a loop as $L_i$, with the subscript distinguishing different loops. Suppose our variational execution generates a trace $t_v$. Our static partial ordering guarantees that $t_v$ has the following properties:

- If a loop $L_i$ is executed, VBlocks belonging to $L_i$ are adjacent to each other in $t_v$, without any VBlock that does not belong to $L_i$ in between. We call this region a looping region of $L_i$, denoted as $RV_i$. This can be proven by contradiction. If there is a VBlock $b$ (not belonging to $L_i$) inside $RV_i$, between $b_x$ and $b_y$ (both $b_x$ and $b_y$ are part of the loop $L_i$), $b$ must have the same transitive predecessor relation with $b_x$ and $b_y$, because $b$ is not part of the loop $L_i$. If this is the case, our static partial ordering would require $b$ to either precede both $b_x$ and $b_y$ or fall behind $b_x$ and $b_y$ in the trace. This is contracting to the assumption that $b$ is between $b_x$ and $b_y$ in the trace $t_v$.
- In $t_v$, any VBlock $b$ that is outside looping regions have no duplication. This can also be proven by contradiction. Suppose $b$ (not belonging to any loops) appears twice in $t_v$, there must be a transitive predecessor of $b$ between these two appearances of $b$ in $t_v$ because $b$ does not belong to any loops. However, $b$'s transitive predecessor cannot appear between two occurrences of $b$ in $t_v$, due to our static partial ordering.
- For any loop $L_i$, there is at most one looping region $RV_i$ in $t_v$. Otherwise, $L_i$ must be a inner loop of another bigger loop. If this is the case, there must be at least one conditional jump in the outer loop, and therefore the outer loop fails to satisfy the simple loop premise.

Based on these properties, we have $length(t_v) = \sum_i length(RV_i) + n$, where $length(RV_i)$ denotes the number of elements in $RV_i$, and $n$ denotes the number of VBlocks in $t_v$ that are not part of any loops.

Now if we consider the concrete traces represented by $t_v$, in order to produce the optimal merging of these traces $t_o$, we need to take two steps: (1) merge looping regions of concrete traces and (2) merge VBlocks that do not belong to any loop.

1. From Lemma 1, we know that the length of merging all looping regions of $L_i$ across concrete traces is determined by the longest looping region, which we denote as $length(RMax_i)$.
2. Merging VBlocks that do not belong to any loop would result in $n$ elements. This is equivalent to merging concrete traces of acyclic control-flow graphs, which we have already proven in the first half of this proof.

Thus, we have $length(t_o) = \sum_i length(RMax_i) + n$. For any loop $L_i$, the length of its looping region $RV_i$ in $t_v$ (if exists) is bounded by $length(RMax_i)$, (i.e., $length(RV_i) \leq length(RMax_i)$), as we have proven in Lemma 2. On the other hand, $RV_i$ is guaranteed to represent the longest looping of $L_i$ in concrete traces because the context with the longest looping must be executed to satisfy

correctness. So, $length(RV_i) \geq length(RMax_i)$, which gives us $length(RV_i) = length(RMax_i)$. Since $length(t_v) = length(t_o)$, we achieve optimal sharing.

□

## 3.9 Summary

While variational execution has been applied in different areas, an efficient implementation is still missing for practical use. We proposed to implement variational execution by transforming programs at the bytecode level. Our approach is transparent to the developers, and has various advantages such as making use of underlying optimizations of the JVM and remaining portable to different JVMs. Our approach transforms individual instructions and modifies the control flow of methods to exploit sharing of common execution across configurations. Even with aggressive modification to the control flow decisions, we formally proved that our transformation to the control flow is correct for all cases, and optimal for a large subset of cases. We further optimized our implementation with two different optimizations, each of which optimizes our approach from different aspects. With an empirical evaluation on 7 highly configurable systems, we show that our approach is 2 to 46 times faster while saving up to 3 quarters of memory usage when compared to the state of the art. A monitoring at runtime further confirms that we achieve 99.8% optimality for the methods that we cannot guarantee optimal sharing. Overall, our results indicate that our approach is useful for analyzing highly configurable systems in practice.

The more scalable and efficient variational execution proposed in this chapter allows us to extend existing work on exploring intentional variations, but more importantly, lays the foundation for branching into a new direction of exploring speculative variations. In the next two chapters, we make the first attempt to use variational execution to analyze complex search spaces of two important areas—higher-order mutation testing (Chapter 4) and automatic program repair (Chapter 5). More broadly, we hope that the techniques we used to scale and improve variational execution can inform more scalable and efficient implementations of similar techniques such as symbolic execution.

# Chapter 4

# Higher-Order Mutation Testing

In this chapter, we explore *interactions* among *speculative variations* in mutation testing. In this context, *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.

Higher-order mutation has the potential for improving major drawbacks of traditional first-order mutation, such as by simulating more realistic faults or improving test-optimization techniques. Despite interest in studying promising higher-order mutants, such mutants are difficult to find due to the exponential search space of mutation combinations. State-of-the-art approaches rely on genetic search, which is often incomplete and expensive due to its stochastic nature. First, we propose a novel way of finding a complete set of higher-order mutants by using *variational execution*. Second, we use the identified complete set of higher-order mutants to study their characteristics. Finally, we use the identified characteristics to design and evaluate a new search strategy, independent of variational execution, that is highly effective at finding higher-order mutants even in large codebases.

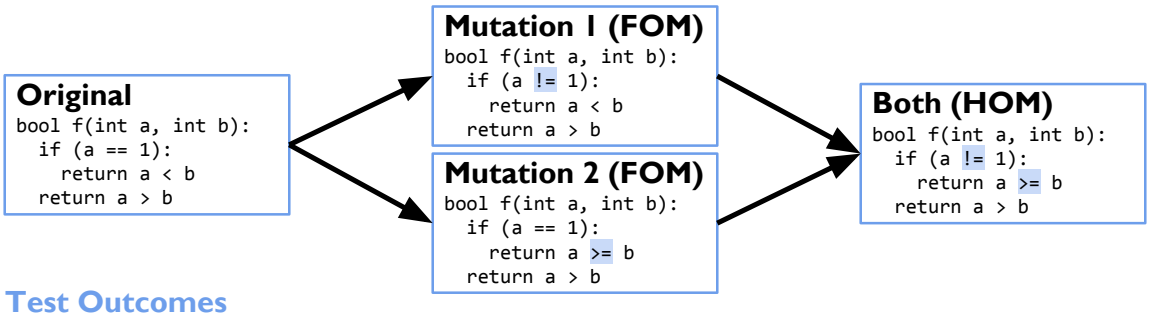
This chapter shares material with the following publication [164]:

- Chu-Pan Wong, Jens Meinicke, Leo Chen, João P. Diniz, Christian Kästner, and Eduardo Figueiredo. 2020. Efficiently finding higher-order mutants. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, 1165–1177.

## 4.1 Strongly Subsuming Higher-Order Mutants

Mutation analysis has been studied for decades in research [125] and is increasingly adopted in industry [129, 130]. Mutation analysis has many applications, including assessing and improving test suite quality, generating or minimizing a test suite, or as a proxy for evaluating other research techniques such as fault localization [62, 125]. Traditionally, mutation analysis injects syntactic mutations into an existing program and runs the existing tests to assess whether the tests are sensitive enough to detect the mutations.

*Higher-order mutation* is the idea of combining multiple mutations to represent more subtle




**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. |
|---|---|---|---|---|---|
| $T_1$: `assert f(1, 2)` | ✓ | ✗ | ✗ | ✗ | $m_1 \vee m_2$ |
| $T_2$: `assert !f(0, 3)` | ✓ | ✗ | ✓ | ✓ | $m_1 \wedge \neg m_2$ |
| $T_3$: `assert !f(1, 1)` | ✓ | ✓ | ✗ | ✓ | $\neg m_1 \wedge m_2$ |

Figure 4.1: Example of mutations with their test outcomes.

changes, more complex changes, or changes that better mirror human mistakes [56]. To that end, Jia and Harman [56] distinguish *first-order mutants*, consisting of a single change, from *higher-order mutants* that combine multiple changes (cf. Fig. 4.1). While most research on mutation analysis has focused on first-order mutants, recent studies claim that higher-order mutants are less likely to be equivalent mutants [74, 99, 104, 126] and that higher-order mutants can reduce test effort [54, 56, 132]. In Section 4.2.1, we will discuss a specific use case with a motivating example.

A key challenge in adopting higher-order mutation is *identifying* beneficial higher-order mutants. Most higher-order mutants are as easy to kill as their constituent first-order mutants, due to coupling. Jia and Harman [56] argue that only a subset of all possible combinations better simulate real faults and increase the subtlety of the seeded faults. Specifically, Jia and Harman [54, 56] look for what they name a *strongly subsuming higher-order mutant (SSHOM)*, a particular kind of higher-order mutant that is harder to detect than its constituent first-order mutants, as we will explain in Section 4.2. However, SSHOMs are tricky to find among the vast quantity of possible combinations of first-order mutants. Current approaches use genetic-search techniques, guided by a simple fitness function [46, 54, 56, 83]. Since SSHOMs are difficult to find, little is known about them and their characteristics.

In this work, we develop a technique that can find a *complete* set of SSHOMs for *given first-order mutations and tests* on *small to medium-sized* programs, which enables us to study

characteristics of SSHOMs. Based on the identified characteristics, we then develop a new heuristic search technique that is lightweight, scalable, and practical. Overall, we proceed in three steps:

*(1) Variational Search:* For the purpose of studying SSHOM in a controlled setting, we develop a new search strategy $\texttt{search}_{var}$ 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 and Chapter 3), a dynamic-analysis technique that jointly explores many similar executions of a program. Conceptually, our approach searches for 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 search as a *Boolean satisfiability problem* to enumerate *all* SSHOMs. An exploration of *all* possible mutant combinations with variational execution is often feasible for *small to medium-sized* programs because variational execution shares commonalities among repetitive executions. Though it does not scale to all programs, analyzing a complete set of SSHOMs for smaller programs and their test suites allows us to study SSHOMs more systematically.

*(2) Characteristics Analysis:* We study the characteristics of the identified higher-order mutants from Step 1. Where previous approaches found only a few samples of higher-order mutants, we have a unique opportunity to study the characteristics of higher-order mutants on a much more complete set. We analyze characteristics, such as the typical number of mutants combined and their distance in the code. This helps us better understand higher-order mutants without the potential sampling bias from a search heuristic. For example, we found that most SSHOMs are composed of fewer than 4 first-order mutants and that constituent first-order mutants tend to locate within the same method or the same class.

*(3) Prioritized Heuristic Search:* Finally, we develop a second new search strategy $\texttt{search}_{pri}$ that prioritizes likely promising combinations of first-order mutants based on the characteristics identified in Step 2. The $\texttt{search}_{pri}$ is easy to implement and does not require the heavyweight variational analysis of $\texttt{search}_{var}$. Although it does not provide any completeness guarantees, it is highly efficient at finding higher-order mutants fast and scales to much larger systems with thousands of first-order mutants. We evaluate the new search strategy using a different set of larger systems to avoid overfitting. Our results indicate that the identified characteristics indeed effectively guide the search. For example, we found 390,533 SSHOMs among combinations of 103,663 first-order mutations, where existing search approaches can barely find any.

We summarize our contributions as follow:

- We propose a novel way of using variational execution to find a *complete* set of SSHOMs for a given set of first-order mutations and tests, by formalizing the search as a Boolean satisfiability problem. An evaluation of small to medium-sized programs shows that we can achieve completeness and simultaneously increase efficiency (Section 4.3).
- Using the identified set of SSHOMs, we make the first step in studying the basic characteristics of SSHOMs to inform future research (Section 4.4).
- To show how useful the characteristics are, we use them to design a new lightweight prioritized search strategy, independent of variational execution. We evaluate the prioritized search strategy on a fresh set of larger benchmarks, showing that the new search is scalable and generalizable (Section 4.5).

## 4.2 Higher-Order Mutants

Mutation analysis introduces a set of syntactic changes to a software artifact and observes whether the previously passing test suite is sensitive enough to detect the changes (termed "to kill the mutant"). Traditionally, many simple small changes are explored in isolation, one at a time; several catalogs of mutation operators that perform small syntactic changes exist [73, 125].

In its simplest form, *higher-order mutants* are combinations of two or more first-order mutants [54, 56]. The set of possible second-order mutants grows quadratically with the size of the set of first-order mutants from which they are combined; if considering combining more than two first-order mutants, the set of possible higher-order mutants grows much faster.

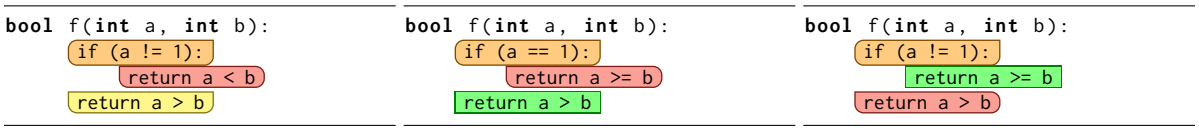
Many higher-order mutants are of little value in practice, because a test that would kill any constituent first-order mutant will likely also kill the higher-order mutant, discussed as the coupling effect hypothesis [120]. However, Jia and Harman [56] argue that there exist several classes of higher-order mutants that exhibit interesting behavior. They specifically highlight *strongly subsuming higher-order mutant (SSHOM)*, in which the constituent mutants interact in ways making the higher-order mutant hard to kill, as we will explain in detail in Section 4.2.2.

### 4.2.1 Usefulness of Higher-Order Mutants

A recent survey of over 39 papers on higher-order mutation testing [41] summarized a large number of different application scenarios for higher-order mutants claimed in prior research, including mutant reduction [40, 46, 52], coupling effect analysis [43, 56], equivalent mutant reduction [75, 99], test data evaluation [45], and test suite reduction [46, 104]. In the following, we illustrate a concrete example of how higher-order mutations can be useful to software-engineering researchers for creating synthetic, but challenging faults to evaluate various software engineering tools.

The effectiveness of many approaches in software-engineering research needs to be evaluated on faults in software systems. For example, fault localization tools need to evaluate how accurately they can localize the faults, test suite generation tools need to evaluate how effective the generated tests are at finding bugs, and program repair tools need to evaluate how many faults they can repair. When evaluating their tools, researchers often have the choice of running evaluations on a curated, often small, set of real bugs or running on large numbers of synthetically seeded bugs. Both approaches have known benefits and drawbacks:

- Seeded faults are convenient: Easy to create and providing a perfect ground truth, they allow researchers to run experiments with very large numbers of faults on almost any system. For example, fault localization techniques were often evaluated on artificially seeded single-edit faults, such as those in the *Siemens* test suite [51] (e.g., [2, 59, 94, 128, 135]). Researchers have been critical of this style of evaluation, arguing that seeded single-edit faults are not representative of most real faults (which often require fixes in multiple locations) [62, 175] and that fault localization techniques may not generalize as they are over-optimized in finding such simple single-edit faults [128].
- In contrast, if curated well, datasets of real faults can be much more representative of realistic usage scenarios. Research on automated program repair is almost exclusively evaluated on a few hundred real faults [89]. For example, the widely used *Defects4J* dataset [61] curated




(a) Mutation 1 (b) Mutation 2 (c) HOM

Figure 4.2: Suspicious lines based on spectrum-based fault localization [59]. The ranking is shown as the intensity of danger, suspicious, caution and safe.

438 faults from 5 libraries. Creating high-quality datasets of realistic and representative faults is challenging and typically requires significant human and engineering effort [61, 98, 154]. Therefore, while it is easy to seed millions of faults in almost any program, only a few datasets of curated real faults are available, often only with moderate numbers of faults in a small number of libraries or programs. Some researchers warn that overly focusing on a few shared datasets of faults leads to approaches that overfit the available faults [33, 154].

In this tension between simple seeded faults and expensive to curate real faults, higher-order mutation may provide a compromise. Certain kinds of higher-order mutants, in particular SSHOMs that we study in this work, are *more subtle* and *harder to kill* (shown both theoretically [43] and empirically [44, 54, 56, 83, 124]). They are more promising to simulate real faults than traditional first-order mutants: For example, Zhong and Su [175] and Just et al. [62] found that more than 50–70 % of real faults are caused by faults in more than two locations. Just et al. [62] also found that 73% of real faults are coupled to mutants, while on average 2 mutants are coupled to a single real fault. That is, certain kinds of higher-order mutants may be more representative of real faults. Thus, assuming we can find them efficiently, which is the goal of this work, we can still automate their creation and seed thousands of these more challenging faults in almost any software system.

Let us illustrate the potential of higher-order mutation for fault localization in Figure 4.2. Our example program from Figure 4.1 is mutated with two first-order mutants, which are later combined to form a higher-order mutant; note how this higher-order mutant fails for fewer test cases than the constituent first-order mutants. In this simple setting, the classic fault localization technique Tarantula [59] works quite well for the first-order mutants, highlighting the mutated lines as shown in Figure 4.2; but Tarantula fails to report the two mutated lines of the higher-order mutant, instead highlighting the unchanged line. This example shows how fault localization fails to locate the faulty lines of interacting mutations, which, as discussed, may be expected for realistic faults [62, 175]. As a further consequence, a program repair technique based on spectrum-based fault localization may not even attempt to fix the first return statement [88].

To realize the full potential of higher-order mutants for these and other use cases, it is critical to have an efficient way of finding interesting higher-order mutants. In this work, we do not reevaluate the usefulness of HOMs for various use cases [41] or how well they represent real faults [56, 62, 175], which has been studied repeatedly and comprehensively in prior work [32]. Instead, we focus on a technical problem that made SSHOMs too costly and impractical: *How to efficiently find SSHOMs.*

### 4.2.2 Strongly Subsuming Higher-Order Mutants (SSHOMs)

Jia and Harman [56] classify higher-order mutants into several kinds, highlighting SSHOMs as useful. For this reason, our work targets SSHOMs, though we expect that it can be generalized to other classes of higher-order mutants. Specifically, Jia and Harman [56] define an SSHOM as a higher-order mutant that can only be killed 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 an 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 an even stronger type of SSHOM, which we call *strict* strongly subsuming higher-order mutant, denoted as *strict-SSHOM*.[1] In other words, there must be at least one test case that kills one of the first-order mutants, 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.

Our (manually constructed) SSHOM in Figure 4.1 illustrates this relation: 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.

### 4.2.3 Finding SSHOMs

SSHOM is defined in terms of test results of *first-order mutants*. All existing search strategies aim to find SSHOMs in terms of a given test suite and a given set of first-order mutants. The search space can be large due to the exponential combinatorial explosion of possible mutant combinations. Since only very few of the combinations are interesting and those are hard to find in a vast search space, higher-order mutation testing has long been considered too expensive.

Jia and Harman [56] explored several search techniques to find SSHOMs, finding that genetic search performs best. We will use their genetic-search strategy, together with a brute-force strategy, as baselines for our evaluations. Although the genetic search has been shown to successfully find SSHOMs, it requires considerable resources to evaluate many candidates, involves significant randomness, and cannot enumerate all SSHOMs in the search space.

It is conceptually possible to define SSHOMs in terms of an idealized test suite that represents all (possibly infinite) possible behaviors in the program. In practice though, all search strategies have to work with existing test suites and whether a combination of first-order mutants is considered an SSHOM is evaluated in terms of a given test suite. Different test suites and

[1] SSHOMs have been defined inconsistently in the literature as subset [46] and strict subset [54, 56]. We inherit the definition of SSHOMs from Harman et al. [46], as it is the most recent work. As we will see in the evaluation, the difference between subset and strict subset is significant, so we make the distinction explicit, introducing strict-SSHOM as a distinct subclass and reporting results for both.

```
def findSSHOMs(program P, mutants M, testsuite T):
  failing_conditions = Map[Test, FailingCondition]()
  # Merge all first-order mutants into one meta-program
  mutated_P = encode_all_mutants(P, M)
  for (test ← T):
    # Variational execution returns under what combinations of mutants the
    # given test fails, compactly represented as a propositional formula.
    f_t = variational_execution(mutated_P, # mutated program
                                test,      # entry point of execution
                                M)         # symbols representing mutants
    failing_conditions.add(test → f_t)
  # search of SSHOMs as a satisfiability problem, using Equation 2-4
  constraint = encodeSAT(failing_conditions, T, M)
  # allSAT returns all solutions to the constraint, each represented
  # as a set of activated variables (first-order mutants)
  found_SSHOMs = allSAT(constraint)
  return foundSSHOMs
```

Figure 4.3: Using variational execution to find SSHOMs.

different first-order mutants may result in different SSHOMs; SSHOMs found with a specific test suite could be interpreted as *approximations* of SSHOMs potentially found with an idealized test suite. A specific test suite and set of first-order mutants form a large but finite search space and it is possible to define and find a *complete* set of SSHOMs *with regard to those given mutants and tests*, as we will discuss in Section 4.3.

In this work, in line with prior work on finding SSHOMs, we focus on finding SSHOMs with regard to a fixed test suite and fixed set first-order mutants, as would be useful in the fault localization and program repair scenarios discussed above. Although orthogonal to the goal of this work, which is improving existing search strategies, in Section 4.6, we discuss the notion of SSHOMs in terms of a theoretical idealized test suite and the influence of test suite size on identifiable SSHOMs.

## 4.3 Step 1: Complete Search With Variational Execution (search$_{var}$)

In this step, we develop search$_{var}$ to compute a *complete* set of SSHOMs with respect to given tests and first-order mutants, so that we can study their properties later. Figure 4.3 shows the pseudo-code of applying variational execution to find SSHOMs.

First, given a program under analysis $P$, we mutate it into $P'$ by applying our mutation operators *exhaustively* at every applicable location to generate all first-order mutants upfront. We represent each first-order mutant as a Boolean option and use a ternary conditional operator to encode the change. Mutations to the same expression are expressed as nested ternary conditional expressions. For example, we show below how we encode the two first-order mutants from Figure 4.1.

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

After encoding first-order mutants, we use variational execution as a black-box technique

to explore, for each test case, under what combinations of first-order mutants the test would fail. In a nutshell, variational execution runs the program by dynamically tracking the differences in program state that are caused by mutations (similar to executing the program symbolically with symbolic values for all mutations) [7, 110, 117, 166]. Conceptually, a single run of variational execution is equivalent to running all combinations of first-order mutations in a brute-force fashion, but it is usually much faster due to the sharing of similar executions at runtime [110, 117, 166]. For each test execution, variational execution will return a failing condition, which is a propositional formula that represents exactly the combinations of mutations for which the test fails. We show examples of failing conditions in 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. Using BDDs or SAT solvers, we can then enumerate all solutions, which correspond directly to all SSHOMs. Although the formulas can be large if we have many first-order mutants and test cases and finding satisfiable assignments is NP-hard, modern SAT solving techniques are scalable enough. Our implementation and data are available on GitHub: https://github.com/poosomooso/SSHOM-Search.

### 4.3.1 Mutant Generation

We generate first-order mutants *exhaustively* and encode them all at once into a metaprogram, which is later used for finding SSHOMs. This compact encoding of mutations defines a finite search space, which is critical for variational execution to be efficient [165]. Similar encodings have been explored in different contexts, such as speeding up mutation testing [63, 100, 156]. Using this encoding, we also ensure a fair comparison with baseline approaches by excluding compilation time and using the same metaprograms.

For our experiments, we implemented 3 mutation operators: (1) *Arithmetic Operator Replacement* (AOR, mutating `+`, `-`, `*`, `/`, `%`) (2) *Relational Operator Replacement* (ROR, mutating `==`, `!=`, `<`, `>`, `<=`, `>=`) and (3) *Logical Connector Replacement* (LCR, mutating `||` and `&&`). These comprise 3 of 5 most well-studied mutation operators [121, 122], excluding two further based on recent insights: (4) *Absolute Value Insertion* (ABS) has been shown to be less useful in practice [129], so we excluded it to avoid a meaninglessly large search space. (5) *Unary Operator Insertion* (UOI) would add many more mutants, most of which are likely equivalent to the ones generated from other mutation operators (e.g., mutating `a+b` to `a+-b` using UOI is equivalent to `a-b` using AOR) [73, 100, 129].

### 4.3.2 Variational Execution

We use variational execution to determine which combinations of mutants fail a test case. The novelty of using variational execution lies in the efficient and complete exploration of all mutants, as opposed to one mutant at a time in traditional search-based approaches. For this work, we use variational execution as a black-box technique. Technical details of variational execution are available in existing literature [7, 110, 117, 166] and previous chapters, such as how it works (Chapter 2) and how it can be implemented (Chapter 3). In this chapter, we only provide intuition as a recap.

Conceptually, variational execution is similar to symbolic execution, in that it executes a program with symbolic Boolean values representing mutants and concrete values for test inputs. Specifically, variational execution performs computations with *conditional values* (Section 2.2), which may represent multiple alternative concrete values. For example, a conditional value $\langle \alpha, 1, -1 \rangle$ indicates that it has the value 1 under $\alpha$, and -1 otherwise. Conditional values can represent a *finite* number of alternative *concrete* values distinguished by propositional conditions over symbolic values (representing mutations). Variational execution computes with conditional values and propagates them along data and control flow. At control-flow decisions, both branches are explored under corresponding symbolic path conditions; afterward, state is merged again into conditional values to exploit sharing in subsequent statements. In a nutshell, variational execution can be considered as an extreme design choice among various forms of symbolic program evaluation [15, 17, 25, 72, 143] for finite domains, in which computations are maximally performed on concrete values, but Boolean symbolic values may distinguish between multiple concrete values in program state [7, 110, 166].

For our purposes, we consider all Boolean options representing first-order mutants as symbolic options. This way, all state changes caused by mutants can be compactly tracked, which enables us to explore all combinations of mutants at the same time. As output, we determine under which combinations of mutants a test case fails (propositional formula over first-order mutants as illustrated in Fig. 4.1), by simply observing under which condition any asserted expression evaluates to *false*.

In theory, mutant interactions can cause a combinatorial explosion in conditional values where exponentially many alternative values for different combinations of mutants need to be tracked for a single variable. However, in practice, not all mutants affect each test and not all mutants interact, enabling an often reasonably efficient exploration of all feasible combinations. We defer the discussion of this scalability issue to Section 4.3.4.

In this work, we use VarexC, the state-of-the-art implementation of variational execution for Java. Implementation details about VarexC are available in Chapter 3.

### 4.3.3 SSHOM Search as a SAT Problem

We use the output of variational execution—propositional formulas indicating under which combinations of mutations each test fails—to construct a single formula that is satisfiable exactly for those assignments that represent SSHOMs, based on our definition of SSHOM in Chapter 4.2.2. This way, the search for SSHOMs is transformed into a Boolean satisfiability problem, which we can solve with BDDs or SAT solvers. To derive the formula, we outline the criteria for identifying SSHOMs as defined by Jia and Harman [56] (also see Section 4.2.2) and construct a logical expression for each criterion.

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 ($f$ is generated with variational execution, e.g., see Figure 4.1). As a 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 test $t$ fails for first-order mutant $m$. To identify SSHOMs, we encode three criteria:

First, we ensure that a mutant combination is killed by at least one test, encoding $T_h \neq \emptyset$ in Formula 4.1 (Section 4.2.2):

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

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

Second, 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 (Section 4.2.2):

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

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

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 [56]. As discussed in Section 4.2.2, 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.

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

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 BDDs to enumerate all satisfiable solutions by default. While constructing BDDs can be expensive, getting a solution from a BDD is fast ($\mathcal{O}(n)$, where $n$ represents the number of Boolean variables [19]). In some rare cases where we cannot construct a BDD due to 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. 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.

### 4.3.4 Limitations

While variational execution and the SAT encoding provide a new strategy to find SSHOMs, this approach comes also with severe restrictions, mostly regarding scalability and engineering limitations inherited from the tools we use, which limits broad applicability in practice (which we address with an alternative strategy in Section 4.5).

**Combinatorial Explosion.** Recent studies show that combinatorial explosion is uncommon for the types of highly-configurable programs analyzed with variational execution in the past [110, 134], mainly because programs are usually written by human developers to have manageable interactions among options. When applied to higher-order mutation testing, we did observe some combinatorial explosion caused by random combinations of first-order mutants. For example, we observed cases where interactions of first-order mutants create more than 15,000 alternative concrete values in one single local variable. We argue that this is the essential complexity of the mutated program, and it would be equally difficult for other approaches to exhaustively explore a complex search space like this. However, it is possible to find efficient search strategies when giving up the completeness goal, as we will show in Section 4.5.

In the evaluation of `search`$_{var}$, we manually removed some problematic first-order mutants and test cases that caused an excessive number of interactions (See Table 4.1). For fairness, we remove these mutants and test cases across all compared approaches.

**Environment Barrier.** Similar to symbolic execution, variational execution needs to handle the environment barrier carefully when interacting with an external runtime environment that is not aware of conditional values or path conditions. This barrier often manifests as I/O or native method calls. As discussed in Section 3.3.2, there are several strategies to mitigate this issue, such as creating model classes for these operations. In our study, only a few tests and mutants triggered problematic environment interactions. While solvable with engineering effort, we consider them noncritical for our goal and removed the problematic tests or mutants after manual inspection.

### 4.3.5 Evaluation

In addition to using `search`$_{var}$ to get a complete set of SSHOMs with regard to given tests and first-order mutants, we compare the 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 [46].

**Subject Systems.** We replicate the setup of the largest previous study on higher-order mutation testing [46]. While we cannot perform an exact replication since we could not obtain the original tools from the authors, not all relevant details and parameters have been published, and some engineering limitations discussed earlier, we still select the same subject systems and reimplement search strategies in our own infrastructure. That is, our results cannot be compared directly against the numbers reported in prior work [46], but we report comparable numbers within a consistent setup.

We use the same four small to medium-sized Java programs, *Monopoly*, *Cli*, *Chess*, and *Validator*, all of which come with good quality test suites that are deemed complete by developers [46]. In addition, we use the *triangle* program commonly used in mutation testing [56]. We report the statistics of our subject systems in Table 4.1 (top), which are comparable to those reported in prior work [46], with slight differences likely caused by different mutation operators used and excluded tests (as discussed in Section 4.3.4).

Table 4.1: Subjects and Found (strict-)SSHOMs; the last three subjects and the *Pri* column are discussed in Section 4.5.

| | | | | | | Found SSHOM (and strict-SSHOM) | | | |
|---|---|---|---|---|---|---|---|---|---|
| Subject | LOC | Tests (%used) | LCov | FOMs (%used) | MutScore | Var | Gen | BF | Pri |
| Validator | 7,563 | 302 (83%) | 54% | 1,941 (97%) | 36% (68%) | $1.34^{*}10^{10}$ (281) | 4,041 (0) | 273 (4) | 36,995 (10) |
| Chess | 4,754 | 847 (84%) | 74% | 956 (26%) | 81% (86%) | 3,268† (216) | 484 (0) | 19 (6) | 16,403 (24) |
| Monopoly | 4,173 | 99 (89%) | 74% | 366 (90%) | 80% (83%) | 818 (43) | 81 (4) | 349 (15) | 817 (43) |
| Cli | 1,585 | 149 (95%) | 92% | 249 (51%) | 71% (81%) | 376 (21) | 309 (18) | 326 (21) | 369 (21) |
| Triangle | 19 | 26 (100%) | 100% | 128 (100%) | 92% (92%) | 965 (6) | 949 (6) | 493 (6) | 965 (6) |
| Ant | 108,622 | 1354 (77%) | 53% | 18,280 (92%) | 57% (94%) | - (-) | 1 (0) | 0 (0) | 44,496 (61) |
| Math | 104,506 | 5177 (79%) | 90% | 103,663 (100%) | 66% (71%) | - (-) | 0 (0) | 0 (0) | 390,533 (2,830) |
| JFreeChart | 90,481 | 2169 (99%) | 59% | 36,307 (99%) | 21% (45%) | - (-) | 0 (0) | 6 (0) | 576,725 (513) |

**LOC** represents lines of code, excluding test code, measured with *sloccount*. **Tests** and **FOMs** report the numbers of test cases and first-order mutants we used in experiments, with the percentages relative to the total numbers in parentheses. **LCov** reports line coverage of the tests used in our experiments. **MutScore** reports the mutation score of all used first-order mutants and the score in parenthesis considers only FOMs that are covered by the tests. **Var**, **Gen**, **BF**, **Pri** denote our approach (Step 1, $\texttt{search}_{var}$), the genetic algorithm ($\texttt{search}_{gen}$), brute force ($\texttt{search}_{bf}$), and our prioritized search (Step 3, $\texttt{search}_{pri}$) respectively. † incomplete results, solutions found with SAT solving within the 12 hours budget.

**Baseline Search Strategies.** We compare our approach against the state-of-the-art genetic algorithm [46, 54, 56] and a naive brute-force search. The brute-force search iterates over all valid higher-order mutants, starting from all pairs, then all triples, and so on until a time limit is reached. The brute-force search serves as a reliable baseline as there is no randomness involved.

We reimplemented the genetic algorithm approach based on the description in Jia et al.'s work [53, 54, 56]. As the exact setup was not available or documented, we leave undocumented parameters at default values. The core of the genetic algorithm is a fitness function for candidate higher-order mutants. Following existing work [53, 54, 56] and using the notations in Equation 4.1, we calculate the fitness as $|T_h| \,/\, |\bigcap_{i \in 1 \ldots n} T_i|$.[2] The intuition is that an SSHOM should fail only for a subset of test cases that kill all its constituent first-order mutants. Thus, we use it as a piece-wise function: a fitness of $(0, 1]$ indicates an SSHOM and $(0, 1)$ a strict-SSHOM, with lower fitness more preferable; a fitness of 0 and larger than 1 indicate potential equivalent mutants and non-SSHOMs, respectively, which are discarded between generations of the genetic algorithm.

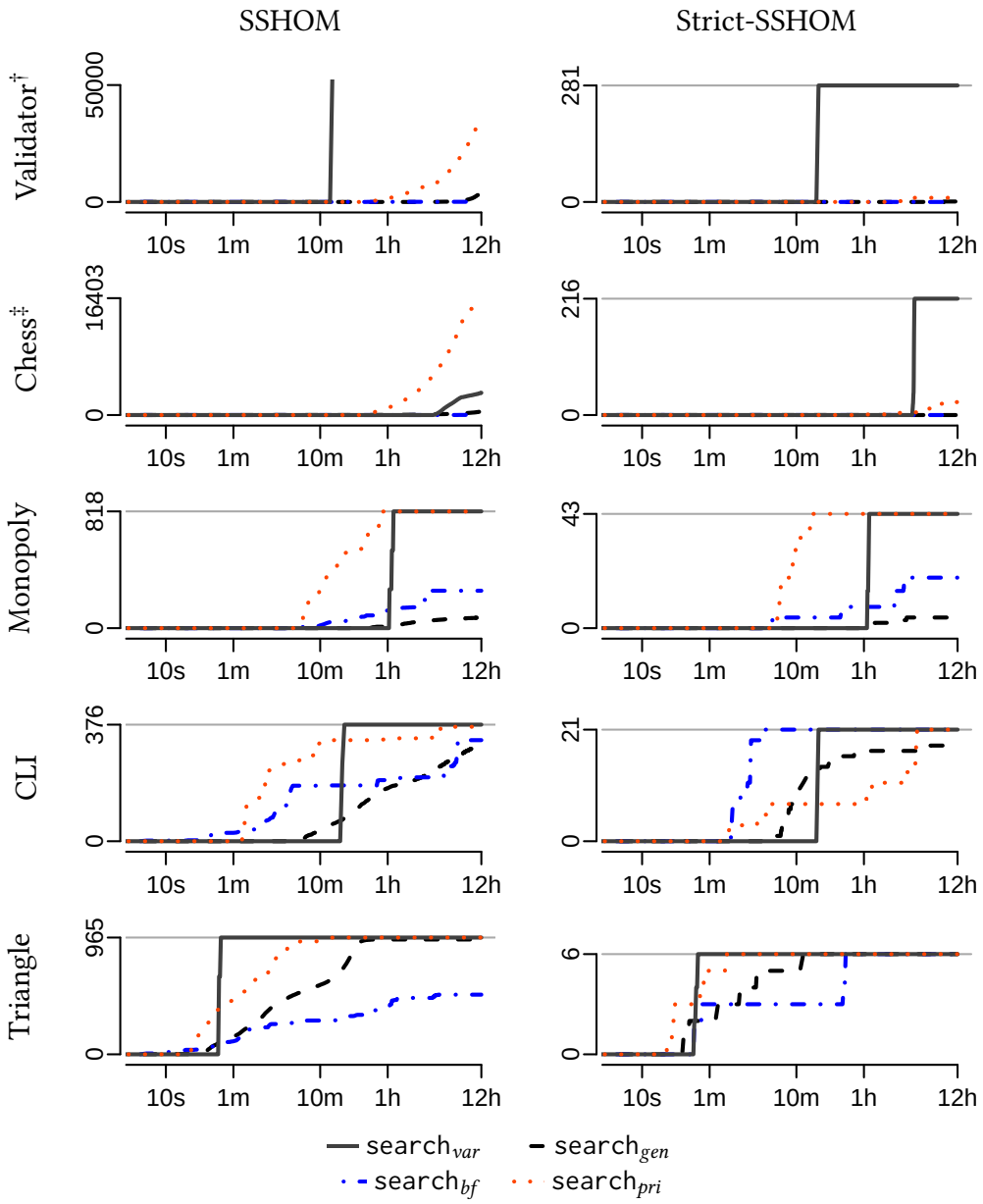
**Measurements.** All experiments were performed on AWS EC2 instances, each of which has an Intel 4-core Xeon CPU with 16GB of RAM. We ran benchmarks to confirm that the performance is stable enough for our measurements across different instances; given that we often demonstrate order-of-magnitude differences in outcomes, differences are unlikely explained by measurement noise. For each search strategy (i.e., search$_{gen}$, search$_{bf}$, search$_{var}$), we measure each subject system three times and report the average, like the three restarts in the work of Harman et al. [46]. We ran each trial of genetic algorithm and brute force for 12 hours.

**Threats to Validity.** External validity might be limited by the specific programs, mutation operators, and test cases. We used common mutation operators and selected subject systems from previous papers to avoid any own sampling bias. From most subject systems, we had to remove some tests or mutations due to technical problems, either engineering limitations of variational execution or issues with memory leaks and infinite loops, which might affect the results to some degree—though we do not expect a systematic bias. Our study only considers three representative mutation operators among all possible ones [125] and may not generalize to other operators. A further analysis of the sensitivity of SSHOMs to a wide array of mutation operators is outside the scope of this work.

Regarding internal validity, like other studies, our results might be affected by possible mistakes in our implementations or measurements and especially by we reimplemented the existing search$_{gen}$ approach. To mitigate this issue, we verified that the SSHOMs found by search$_{gen}$ and search$_{bf}$ are a strict subset of the ones found by search$_{var}$. For SSHOMs found only by our approach, we additionally verified a sample manually to ensure they are SSHOMs.

To reduce the impact of nondeterminism in performance measurements and genetic search, we report averages across 3 runs, as in previous work [46]. Most differences are large, far exceeding the margins of error from nondeterminism or measurement noise.

[2]The fitness function has been defined either using intersect of $T_i$ [53] or union [54, 56]. We use the former in our reimplementation as it more precisely captures our intuition of SSHOMs.




† 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 4.4: (Strict-)SSHOMs 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.

**Results.** In Table 4.1, we report the number of (strict-)SSHOMs found within the 12-hour time budget and, in Figure 4.4, we plot the numbers of (strict-)SSHOMs found over time. Note that by construction, if $\texttt{search}_{var}$ 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 $\texttt{search}_{var}$ this upper bound would not be known.

These results show clear trends: $\texttt{search}_{var}$ 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 very 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 [110, 117, 166].

In contrast, $\texttt{search}_{gen}$ and $\texttt{search}_{bf}$ can test many candidate SSHOMs before variational execution terminates and 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, $\texttt{search}_{bf}$ is fairly effective as it systematically prioritizes pairwise combinations which are more common among SSHOMs than combinations of more than two mutants, as we will discuss.

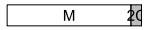
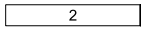
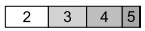
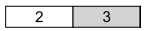
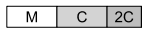
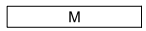
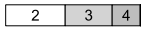
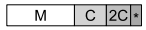
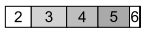
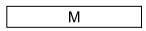
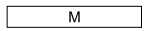
In summary, for systems where variational execution scales, $\texttt{search}_{var}$ can find *all* SSHOMs whereas other approaches find only an often much smaller subset within a 12h time window. Whereas prior approaches often find their first SSHOMs faster, $\texttt{search}_{var}$ needs more time upfront for variational execution but can then enumerate SSHOMs very quickly. To scale $\texttt{search}_{var}$ to more realistic programs, more engineering is needed to overcome the limitations discussed in Section 4.3.4. Nevertheless, $\texttt{search}_{var}$ is valuable to the research community as it provides a precise and efficient way of identifying all SSHOMs.

## 4.4 Step 2: SSHOM Characteristics

In this 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. [79] 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. We started by randomly sampling a large number of identified SSHOMs (among the pool of all SSHOMs). We manually inspected the sampled SSHOMs to pose hypotheses about common characteristics. We then operationalized the hypothesized characteristics (i.e., develop measures to apply across all SSHOMs) to quantitatively validate them. We repeated the process until we could not identify additional hypotheses. Due to space constraints, we only report

Table 4.2: Characteristics of SSHOMs and strict-SSHOMs found in our subject systems.

| | Order | | Equal-Fail Rule | | N+1 Rule | | Distribution | |
|---|---|---|---|---|---|---|---|---|
| Subject | SSHOM | strict-SSHOM | SSHOM | strict-SSHOM | SSHOM | strict-SSHOM | SSHOM | strict-SSHOM |
| Validator† | - |  | - | 96% | - | 99% | - |  |
| Chess† | - |  | - | 76% | - | 38% | - |  |
| Monopoly |  |  | 11% | 100% | 99% | 100% |  |  |
| Cli |  |  | 53% | 5% | 98% | 100% |  |  |
| Triangle |  |  | 8% | 17% | 98% | 50% |  |  |

Order counts the number of constituent first-order mutants; equal-fail and N+1 rule explained in text; distribution: all constituent first-order mutants in the same method (M), multiple methods in the same class (C), two classes (2C), or spread across more than two classes (*).

† for Validator and Chess we omit statistics because we cannot enumerate all possible SSHOMs (too many in Validator and incomplete set in Chess)

characteristics for which we could quantitatively identify strong support.

**Mutation Order.** SSHOMs and strict-SSHOMs are typically composed of only very 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 very few first-order mutants. Although we found a few SSHOMs that are up to sixth-order in Chess and Triangle, such cases are rare, especially for strict-SSHOMs. We plot the distribution of orders for both *SSHOMs* and *strict-SSHOMs* in Table 4.2.

**Equivalent Test Failures.** In multiple subject systems, many SSHOMs and strict-SSHOMs are composed of first-order mutants that are killed by the same set of test cases (nonstrict-SSHOMs are often killed by the same test cases, whereas strict-SSHOMs necessarily are killed by fewer). In Table 4.2, we report how many of the SSHOMs and strict-SSHOMs in each project could be found when only combining first-order mutants that are killed by the same test cases, which we name *Equal-Fail SSHOMs*.

**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 forms 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. In Table 4.2, we report how many of the (strict-)SSHOMs in each project with more than two constituent first-order mutants could be generated with such a rule.

**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 [56]. We plot the distributions for all subject systems in Table 4.2.

**Other.** We also explored other patterns that may inform search heuristics, such as common combinations of mutation operators (using frequent-itemset mining [3]), but found no additional strong patterns. While we believe 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.

## 4.5 Step 3: Prioritized Search (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 search$_{var}$ strategy.

### 4.5.1 Search 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.

Conceptually, we calculate a penalty for every candidate higher-order mutant and prioritize those candidates with the lowest penalty. We compute the weighted sum of three factors:

$$penalty = \omega_1 \cdot order + \omega_2 \cdot testDiff - \omega_3 \cdot isN1 \tag{4.5}$$

First, we assign penalties based on the number of constituent first-order mutants (*order*): a candidate with a higher order receives a larger penalty than a lower-order candidate, thus, prioritizing candidates with lower order that, as our data shows, are more likely to be SSHOMs. Second, we penalize candidates constructed from first-order mutants that do not get killed by the same test cases (*testDiff*, counting the number of test cases that can kill only a subset of all constituent first order mutants), generalizing our *Equivalent Test Failures* insight: if all first-order mutants are killed by the same test cases, the candidate is likely to be an SSHOM, and thus gets a 0 penalty, whereas mutants that are killed by different test cases are less likely to form an SSHOM, and thus is deferred with a higher penalty. Finally, we reduce the penalty of a candidate if the *N+1 Rule* applies (*isN1*, returning 1 or 0); that is, if a candidate can be constructed by adding one more first-order mutant to a known SSHOM, the candidate receives a boost and gets prioritized. By default and for our evaluation, we assign the weights $\omega_1 = 5$, $\omega_2 = 1$, and $\omega_3 = 15$, based on our experience with the subject systems in Section 4.3.5.

Unlike previously used genetic search strategies, where the exploration order nondeterministically depends on random mutation and crossover in every generation, $\texttt{search}_{pri}$ explores candidates in a deterministic order (lexical order if two candidates have the same priority).

### 4.5.2 Implementation

Since we cannot enumerate and sort all possible candidate higher-order mutants for large programs, and even the execution of all first-order mutants may take a long time, we devise an algorithm for $\texttt{search}_{pri}$ that identifies likely candidates in batches, shown in Figure 4.5. In each batch (configurable, by default one Java package at a time), we enumerate all candidate higher-order mutants up to a distance and order bound, then sort these candidates by priority, and finally explore these candidates in order until a (time) budget is reached for that batch. Batching and bounding the search is feasible since the order and distribution characteristics dominate the prioritization anyway and candidates beyond those bounds would be explored only very late. If needed batches could be revisited later with larger bounds to explore more (less likely) candidates.

```
def findSSHOMs(program P, mutants M, testsuite T,
               maxOrder, maxDist, budget):
  foundSSHOMs = ∅
  # explore the program one fragment at a time
  for (batch ← fragments(P)):
    # identify reachable first-order mutants in fragment
    mutants = reachable(M, batch)
    # run tests on reachable first-order mutants
    fomTestResults = for (m ← mutants) evaluate(T, {m})

    # enumerate candidate SSHOMs up to order and distance bounds
    candidates = enumerateCandidates(mutants, maxOrder, maxDist)
    # compute priorities for each candidate
    priorities = computePriorities(candidates, fomTestResults, {})

    # explore candidates in decreasing priority
    while (candidates ≠ ∅ ∧ within budget):
      candidate = getNext(candidates, priorities)
      candidates -= candidate
      homTestResult = evaluate(T, candidate)
      if (isSSHOM(fomTestResults, homTestResult)):
        foundSSHOMs += candidate
        # update priorities based on N+1 rule
        priorities = computePriorities(candidates, fomTestResults,
                                       foundSSHOMs)
  return foundSSHOMs
```

Figure 4.5: Characteristics-based prioritized search algorithm.

After batching, our algorithm identifies all first-order mutants defined within the given batch (function `reachable`) and runs the test suite for each of these first-order mutants to identify which tests fail (function `evaluate`). Subsequently, the algorithm enumerates all candidates (function `enumerateCandidates`) up to a given order bound (by default, mutants composed of up to 6 first-order mutants) and up to a given distance bound (by default, up to 4 methods spread across at most 3 classes). We also discard candidates where constituent first-order mutants have no common failing tests because they cannot form SSHOMs according to the definition. Having a manageable set of candidates in the given batch, the algorithm computes priorities (function `computePriorities`) for all candidates using Equation 4.5 and then explores these candidates in order of decreasing priorities (function `getNext`) until either all candidates are explored or a (time) budget has been reached in that batch (by default, 1 hour per batch). For each candidate, it runs the test suite and compares test results to determine whether a (strict-)SSHOM has been found (function `isSSHOM`); identified SSHOMs are collected and used to recompute priorities based on additional information for the N+1 rule.

### 4.5.3 Evaluation

We evaluate how *effective* `search`$_{pri}$ is at finding (strict-)SSHOMs, and additionally evaluate how it *generalizes* and *scales* to much larger systems than in prior studies on SSHOMs (and used in Section 4.3.5).

**Subject Systems.** We evaluate `search`$_{pri}$ both on the subjects previously used in Section 4.3.5 and on a fresh set of much larger subject systems. The comparison against the 5 previously

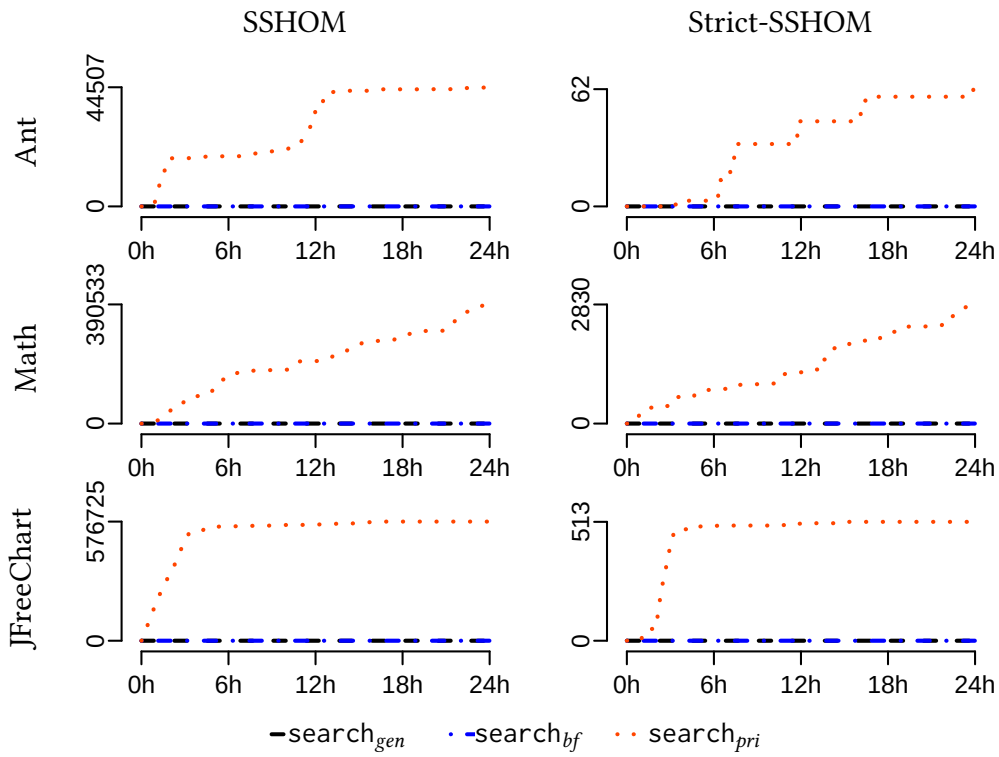



Figure 4.6: (Strict-)SSHOMs found over time, averaged over 3 executions. Note that time is plotted in a linear scale as SSHOMs are found consistently over time due to batching.

used subject systems allows us to compare effectiveness against the ground truth derived from variational execution, but the results may suffer from overfitting, as we evaluate the search strategy on systems from which the insights that drive its design have been derived.

Hence, we use 3 additional subjects, 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. To select the new subject systems, we collected all research papers published in the last 5 years at ASE, FSE, and ICSE that have the word "mutation" or "mutant" in the title. We then selected the five largest Java systems used, discarding two for which we failed to reliably execute the tests. We did not run $\texttt{search}_{var}$ on these systems, but we still had to exclude some tests or mutants (reported in Table 4.1), due to technical issues like hard-to-terminate infinite loops.

**Measurements.** We mirror our previous setup in Section 4.3.5 and count the number of (strict-)SSHOMs found over time. We collect measurements for $\texttt{search}_{bf}$, $\texttt{search}_{gen}$, and $\texttt{search}_{pri}$. Experiments on the small subject systems were performed on the same AWS EC2 instances (Section 4.3.5). For the new systems, we collected measurements on Linux machines with 1.30GHz Intel i5 CPU and 16 GB of memory. When using $\texttt{search}_{pri}$, we used batching for the new larger subject systems, one package at a time, with a 1 hour budget for each package; all other parameters were left at their defaults (described above). For the new subject systems, we ran each measurement for 24 hours, repeated $\texttt{search}_{gen}$ 3 times.

All considered search strategies require executing the test suite repeatedly for each candidate SSHOM. For the larger systems, long test-execution times severely limit the number of mutants we can explore. To minimize the slowdown from test execution that affects all approaches equally, we implement a standard regression test selection technique [125] that only executes test cases that can reach the candidate mutant (technically, we instrument the program to record which test reaches the location of each first-order mutant and only execute tests that reach at least one first-order mutant of a candidate higher-order mutant). We apply this test optimization for all search strategies.

**Threats to Validity.** In addition to the threats discussed in Section 4.3.5, it would be possible to improve $\texttt{search}_{bf}$ and $\texttt{search}_{gen}$ by applying insights from our research, such as a similar batching strategy to explore one Java package at a time and possibly also other insights from analyzing SSHOM characteristics. When using batching (results not shown), these approaches indeed perform better on the large subject systems but are still significantly outperformed by $\texttt{search}_{pri}$, as we will see subsequently. For brevity and consistency, we only compare $\texttt{search}_{pri}$ against vanilla $\texttt{search}_{bf}$ and $\texttt{search}_{gen}$.

**Results.** On the small subject systems, as shown in Table 4.1 and Figure 4.4, our new search strategy $\texttt{search}_{pri}$ is often very effective, performing at least as well as and usually significantly outperforming both $\texttt{search}_{bf}$ and $\texttt{search}_{gen}$ in all subjects. In a few cases, it even outperforms $\texttt{search}_{var}$: 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 in Table 4.1 and Figure 4.6 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, $\texttt{search}_{pri}$ 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 reasonable precision for finding actual SSHOMs among the tested candidates (60.9 % in *Math*, 29.4 % in *Ant*, and 77.8 % in *JFreeChart*).

We conclude that $\texttt{search}_{pri}$ 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.

## 4.6 Test Suite Relevance

As discussed in Section 4.2.3, for all practical search strategies, SSHOMs are identified in terms of used first-order mutants and tests. Different test suites may result in different SSHOMs, though it is conceptually possible to define SSHOMs in terms of an idealized test suite that covers all (possibly infinitely many) executions of a program.

To explore the influence of the test suite, in this appendix, we explore how different (real and ideal) test suites affect the number of (strict-)SSHOMs in a simple program. Given the large

number of executions involved and the use of symbolic execution to represent the ideal test suite, we limit our exploration to our smallest subject system, the triangle program.

### Real Test Suite of Varying Size

Keeping the first-order mutants fixed, we use $\texttt{search}_{var}$ to compute the set of SSHOMs with regard to different test suites. Instead of the 26 tests used previously, we generate and use a much larger set of 1334 tests for this experiment, by systematically exploring combinations of different inputs for the program: For each of the triangle's three inputs, we consider values from −5 to 5, thus yielding $11^3$ tests (easily reaching 100% line and branch coverage). To achieve maximum first-order mutation score, we manually add 3 more tests and verify that the remaining 3 out of 128 first-order mutants are equivalent mutants. From this large test suite, we downsample test suites of different sizes, each time randomly picking a subset of tests; we then identify (strict-)SSHOMs for this test suite with $\texttt{search}_{var}$. We report the median of five executions to account for randomness in selecting tests.

From Figure 4.7, we can see that the number of SSHOMs heavily depends on the test suite used, as expected (see Section 4.2). Also, there is a clear trend that the number of SSHOMs decreases as we use more comprehensive tests: the more tests used, the more constrained the search becomes and the fewer SSHOMs remain. In contrast, the number of strict-SSHOMs is low across different test suites, likely due to the fact that they are rare.

### Idealized Test Suite Using Symbolic Execution

Since the number of SSHOMs decreases as more comprehensive tests are used, one could expect that the number of SSHOMs will converge if we consider all possible tests (usually infinitely many). Given a set of first-order mutants, one can consider the *SSHOMs with regard to a given test suite* as an *approximation* of a true set of *SSHOMs with regard to an idealized test suite that includes all possible tests*, similar to how comprehensive tests can be used to approximate dominator mutants [64]. One would expect that larger test suites are better approximations of such an idealized test suite. While such a test suite usually does not and cannot exist, for the triangle program we can actually simulate the idealized test suite through symbolic execution. The program is simple enough that formal verification of whether two (mutated) variants are semantically equivalent is decidable and automatable.

To identify SSHOMs with an idealized test suite, we symbolically execute the triangle program with three symbolic variables for the three inputs to compute a symbolic representation of the program output—for which we developed a custom symbolic execution engine. The (possibly infinite) set of tests that distinguishes two programs $p_1$ and $p_2$ is the set of values for which $\phi(p_1) \neq \phi(p_2)$ where $\phi$ computes the symbolic output of the triangle program; if there are no such assignments to the three symbolic inputs, the two programs must be equivalent (as determined by an SMT solver). To check the behavior of a mutation $m$, we use $\Delta(m) = \big(\phi(p) \neq \phi(p+m)\big)$ to denote the symbolic expression that represents all tests that kill the mutant.

Using symbolic execution and an SMT solver, we can now determine whether a higher-order mutant is an SSHOM with regard to an idealized test suite, by using an SMT solver to solve

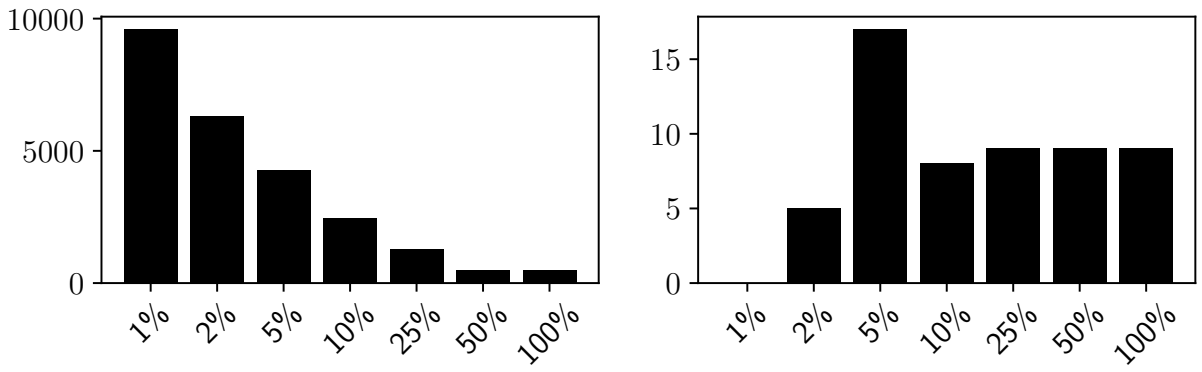



Figure 4.7: Number of found SSHOMs (left) and strict-SSHOMs (right) using different percentages of tests.

constraints that encode Equation 4.1 to determine whether there are any assignments to the symbolic input variables, such that (1) the higher-order mutant fails at least for some tests (i.e., $SAT(\Delta(h))$, (2) that the higher-order mutant fails for those tests where all first-order mutants fail (i.e., $TAUT(\Delta(h) \Rightarrow \bigwedge_i \Delta(m_i))$), and (for strict-SSHOMs) that at least one test input passes for the higher-order mutant but fails for all first-order mutants (i.e., $SAT(\neg\Delta(h) \wedge \bigwedge_i \Delta(m_i))$).

In theory, we can use our symbolic analysis to enumerate and verify all valid higher-order mutants in triangle to establish the set of SSHOMs wrt. the idealized test suite. However, we limit this experiment to only combinations of two and three mutants due to the vast search space. Using the idealized test suite of all possible tests, we found 159 second-order and 157 third-order SSHOMs, and 5 and 3 of them are strict-SSHOMs. Interestingly, all 159 and 157 SSHOMs and 2 (of 5) and all 3 strict-SSHOMs were also identified by $\texttt{search}_{var}$ using the original 26 test cases (Section 4.3). Furthermore, 467 of the 965 SSHOMs and 5 of the 6 strict-SSHOMs identified using the 26 tests in Section 4.3.5 are valid (strict-)SSHOMs with regard to the idealized test suite. That is, *SSHOMs with regard to a given test suite* can indeed be seen as an *approximation* of a true set of *SSHOMs with regard to an idealized test suite.*

## 4.7 Related Work

In this section, we focus our discussions on higher-order mutation testing and refer interested readers to a detailed survey for recent advances in mutation testing in general [125].

**Approaches for Finding SSHOMs.** Early work has investigated different strategies to combine first-order mutants into second-order mutants [74, 99, 104]. Jia and Harman extended this effort to even higher orders using heuristic search looking for certain kinds of valuable higher-order mutants, specifically SSHOMs. They compare a greedy, a hill-climbing, and a genetic algorithm and found that genetic search produces the best results for finding SSHOMs [54, 56]. Since then, higher-order mutation testing has been implemented in different mutation testing tools and frameworks, for different languages [55, 81, 85, 101, 123, 153], usually using some form of heuristic search [54, 56, 83, 124]. Although this work specifically targets SSHOMs, our

approach can be generalized to other types of interesting mutants, by updating the way we encode the search as a Boolean satisfiability problem.

Orthogonal to SSHOMs, researchers have recently investigated an interesting type of hard-to-kill mutants called *dominator mutants* [77, 78]. This line of work searches for the hardest-to-kill first-order mutants among a given set by comparing executions with regard to a test suite. Dominator mutants have been shown to be an effective research tool to study existing mutation testing techniques, for example for gauging mutation test completeness [80] and evaluating selective mutation [79]. Just et al. [64] show that program context can be used to approximate dominator mutants, which might also be promising for future search strategies for SSHOMs.

**Characteristics of SSHOMs.** Although SSHOMs are considered to be one of the most valuable types of higher-order mutants [56], characteristics of SSHOMs have not been systematically studied in the past. Existing work on SSHOMs mostly discusses the quantity of SSHOMs and the difficulty of finding them [46, 54, 55, 56, 83]. For example, Harman et al. [46] discussed how SSHOMs relate to their constituent first-order mutants, but their discussion focuses mainly on test effectiveness and efficiency. Jia and Harman [56] discussed characteristics of a single SSHOM in the Triangle program (also used in our study) but did not explore SSHOM characteristics further. In our work, we can find a complete set of SSHOMs with regard to used tests and first-order mutants, which provides us more data to study what they look like. We hope that future research on generating SSHOMs efficiently can be informed by our work.

**Using Variational Execution.** With regard to using variational execution for mutation testing, Devroey et al. [30]'s work is conceptually closest to our work in that they pursue a complete exploration strategy with similarities to lazy configuration exploration in SPLat [70, 110]. However, they explore only traces in state machines without any joining and thus forgo much possible sharing. Their analysis does not distinguish first-order from higher-order mutants and does not identify or analyze SSHOM. Orthogonal to our work, researchers have also used various techniques to speed up traditional mutation testing, such as sampling tests for mutant executions, condensing mutations into a metaprogram, and using advanced execution sharing techniques [57, 60, 125, 159]. At a technical level, Wang et al. [159]'s work is closest to our work in that they look for possible redundant mutant executions by inspecting program state, but forgo potential joining after splitting mutant executions. Since our main goal of using variational execution is to explore the interactions of first-order mutants rather than speed up mutation analysis, we did not perform a performance comparison.

## 4.8 Summary

To efficiently find SSHOMs, we proceeded in three steps. First, we used variational execution to find *all* SSHOMs in small to medium-sized programs. Second, we analyzed the basic characteristics of the identified SSHOMs. Finally, we derived a new prioritized search strategy based on the characteristics. The prioritized search scales to large systems and is effective (albeit not complete) at finding SSHOMs and outperforms the existing state-of-the-art strategy by far. We

hope that the insights and search strategies from this work can support future work in mutation testing.

The work in this chapter illustrates that variational execution can efficiently navigate the search space of many speculative variations (i.e., first-order mutations) and uncover many interesting interactions among them (i.e., SSHOMs). It also presents a typical workflow of applying variational execution: First, we used variational execution as a black-box technique to *systematically* and *completely* explore the relatively small search spaces of programs that variational execution can easily analyze without too much engineering effort. Next, we analyzed these results to identify interesting findings or characteristics of the search space, which are more likely to generalize because they are derived from a *complete* exploration of the search space instead of a *ad-hoc* or potentially biased exploration as performed in most prior work. Finally, we realized the potential of these findings to improve or design new search strategies that are more effective and more scalable, independent of variational execution. We suspect that a similar workflow could be useful for other heavyweight techniques such as symbolic execution and model checking, by maintaining a reasonable balance between scalability and engineering effort.

In Chapter 5, we will switch focus to another search problem of speculative variations—automatic program repair.

# Chapter 5

# Automatic Program Repair

In this chapter, we explore speculative variations in automatic program repair. In this context, speculative variations are automatically generated edits to a given buggy program. Edits can be small, such as tweaking operators in expressions, but can also be big, such as replacing an entire loop, which poses more challenges to exploring the search space systematically. Interactions of edits yield multi-edit patches and potentially high-quality patches, patches of both kinds remain challenging to identify within a large search space.

The problem of automatically repairing a bugging program is essentially a search problem, in which code transformations are sought to meet a search goal such as passing all the tests. Various search strategies have been explored, but they either navigate the search space in an *ad hoc* way using heuristics, or systemically but at the cost of *limited edit expressiveness* in the kinds of supported program edits. In this work, we explore the possibility of *systematically* navigating the search space without sacrificing *edit expressiveness*. The key enabler of this exploration is variational execution, a dynamic analysis technique that has been shown to be effective at navigating large search spaces.

## 5.1 Automatic Program Repair

Repairing buggy programs is a difficult, time-consuming, and expensive process [136]. A recent report estimates that, in 2017, software failures have affected 3.6 billion people and caused $1.7 trillion financial loss [1]. To reduce the cost of fixing bugs, researchers are working on techniques that automate the program repair process, to find patches to a given bug with little human intervention. Not only have such techniques received lots of research attention in recent years [113], but they also are beginning to see adoption in industrial settings [9, 102].

In essence, program repair techniques generate patches for a given buggy program to satisfy a given specification of program behavior. Specifications can take different forms [113], but this work, as most existing work, targets test-based automatic program repair [95, 97, 107, 162, 163]: Given a program and its test suite with at least one failing test, our approach creates patches that make the whole test suite pass.

Existing approaches to automatic program repair essentially solve a search problem. The search space comprises different program *edits* that could possibly fix the buggy program, such

as copying existing code fragments in the program [162, 163] or modifying operators in `if` conditionals [34, 95, 107]. The *search space* defined by the set of possible edits is large and it grows exponentially if combinations of multiple edits are considered. To navigate such a huge search space, different search strategies have been explored, such as genetic programming [162], guided search using statistical models [97, 163], and formal program synthesis [34, 86, 107]. The keys to solving the search problem of program repair are twofold: *edit expressiveness* and *search effectiveness*, which determine what fixing ingredients are in the search space and how effective the search is at identifying patches that are available in the search space, respectively.

Despite years of research, there is a tension between *edit expressiveness* and *search effectiveness*. This tension essentially divides most existing approaches into two families, namely *heuristics-based* and *semantics-based*:

**Heuristics-Based.** Heuristics-based approaches excel at *edit expressiveness* in that different forms of edits at different AST levels (e.g., expressions, statements) can be explored easily. For example, in terms of edits explored, GenProg is one of the most generic approaches because it can append, replace, and delete different kinds of *statements* [162]. At the other end of the spectrum we have approaches like PAR where only highly specific edit templates are used [71]. Strong edit expressiveness makes it possible to generate patches of different forms, but at the same time inevitably yields a large search space of potential fixing ingredients [96]. The large search space demands an effective way of exploring it, but search effectiveness of heuristics-based approaches is often limited by expensive validation of patch candidates. After a patch candidate is generated by applying certain edit(s) to the buggy program, it is validated by executing the original tests or its subset. The cost of repeated test executions can add up quickly and eventually dominate the entire repair process. To mitigate expensive validation, different heuristics have been explored to prioritize more-likely patch candidates, such as using genetic programming [162] and statistical models [97, 163] to guide the selection of patch candidates for validation. Nonetheless, finding high-quality patches in a large search space of fixing ingredients remains challenging because only a (more or less random) subset of them can be explored. In fact, they often stop at the first identified patch that passes all tests, even though that patch may overfit the test suite, or even when other better patches could have been found in the search space. Finding multi-edit patches is theoretically possible, but unlikely in practice because the search space becomes exponentially larger. HERCULES can generate multi-edit patches effectively, but targets a specific form of multi-edit patches, those that require all edits of a multi-edit patch to be the same or similar [139].

**Semantics-Based.** Semantics-based approaches excel at *search effectiveness*, but at the cost of limited *edit expressiveness*. Semantics-based approaches use *symbolic execution* to encode the given test suite as constraints, which are then fed to program synthesis to *synthesize* patches. Program synthesis uses classic AI search (e.g., SAT, SMT) [137] to effectively search large spaces (e.g., by abstracting and pruning infeasible parts quickly), rendering semantics-based approaches efficient at finding patches. However, existing program synthesis techniques tend not to scale for large synthesis problems, hence the search space is typically constrained. Existing semantics-based approaches only modify *expressions* in conditions or assignments [34, 86, 107,

144, 170]. Moreover, these approaches exclusively synthesize code fragments of boolean and integer types, mainly because of the limited capacity of the underlying constraint solvers [86]. Due to such limited edit expressiveness, patches that require structural changes (e.g., moving statements) or reasoning about other common data types (e.g., floating point and string) are out of scope for existing semantics-based approaches. Multi-edit patch generation can, in theory, benefit from the high search effectiveness of semantics-based approaches, but in practice often limited by the scalability of existing *symbolic execution* and *program synthesis*. For example, DirectFix needs to symbolically execute the whole program under repair to synthesize edits at multiple locations [106]. Angelix mitigates this issue by applying symbolic execution exclusively to a few suspicious expressions, but requires buggy locations to be physically close [107]. S3 essentially repairs each buggy location separately using program synthesis, posing more scalability challenge to the underlying program synthesizer [86].

In this work, we explore the possibility of achieving high *search effectiveness* without sacrificing *edit expressiveness*. Our goal is to perform an efficient systematic search, similar to semantics-based approaches, in a search space of many different kinds of edits, similar to heuristics-based approaches. Our key insight is that a systematic search over many edits and their combinations may be feasible more efficiently because many test executions will exhibit very similar traces and those similarities can be exploited to speed up the search. The similarity of test executions across multiple patch candidates (where edits tend to be focused in a relatively small part of the trace as narrowed down by fault localization techniques) can be leveraged to speed up executions of tests. Moreover, the dynamic information obtained from test executions can potentially provide useful insights for improving patch quality, similar to how similarities of test executions are used to classify correct patches [168]. To exploit test execution similarity, we use variational execution [110, 117, 166] (Chapter 2 and Chapter 3).

As discussed throughout this thesis, variational execution is a dynamic analysis that executes common parts of test executions only once, such as code that is irrelevant to any edits, before or after the edits. Reminiscent of many model checking strategies, when an edit is encountered, variational execution *splits* the execution to compute the program states with and without the edit, but also, importantly, *merges* executions again to execute the rest only once. Conceptually, given a search space of edits as fixing ingredients, a single run of variational execution is equivalent to running all edits and their combinations in isolation. But by splitting and merging executions, variational execution can explore the search space efficiently, as shown in recent work on testing highly configurable systems [110, 117], tracking sensitive information flow [8, 171], and finding higher-order mutants (Chapter 4).

The efficiency of variational execution depends on how individual edits behave. If all individual edits influence program states independently without any interactions (e.g., modifying completely different variables), variational execution can be very efficient. Interactions of individual edits can slow down variational execution because we need to compute alternative program states for all edits and their combinations (but likely much sharing remains to outperform brute force). Whether variational execution can tame the search space of program repair remains an open question.

Our approach also has potential to improve *patch quality*. Similar to semantics-based approaches, our approach analyzes large search spaces effectively and finds many or all plausible patches within that search space. Having access to multiple plausible patches allows us to

deliberately prioritize patches that are more likely to generalize beyond the given test suite (e.g., prioritizing smaller patches, patches that affect less program state, or patches that have more local effects). Furthermore, variational execution can easily collect runtime information that is otherwise only obtainable by expensive alignment and comparison of execution traces, which provides useful insights into control-flow and state changes caused by edits, that are useful for selecting better patches among many plausible patches. In this work, we develop heuristics to rank promising patches, using information of their control-flow and data-flow changes.

We evaluate the feasibility of variational execution using IntroClassJava, a dataset that contains 297 bugs that are small but notoriously difficult to fix. Results show that we can patch 107 of them, including 40 bugs that are exclusively repaired by our approach. For the patched IntroClassJava bugs, we can generate up to 33017 plausible patches (i.e., patches that pass all tests), and up to 32841 correct patches (i.e., patches that meet the specification of expected program behavior). Due to a systematic search, we can find up to hundreds of multi-edit patches. Finally, using runtime information obtained from variational execution, our patch ranking is effective in distinguishing correct patches from plausible but incorrect patches, ranking the correct patches to top 10 for 23 out of 24 evaluated bugs.

To further evaluate scalability, we use two largest subjects from the Defects4J dataset. An evaluation with 282 bugs from Math and Closure shows that our proposed techniques can solve large repair problems in practice, fixing 35 of them with patches of varying quality. Most findings for IntroClassJava can be generalized to Defects4J, despite searching in much large search spaces and running more tests, suggesting that our approach is scalable.

We make the following contributions in this work:

- An exploration of applying variational execution to systematically explore large search spaces of program repair.
- A prototype repair tool built on top of GenProg and VarexC.
- An thorough evaluation showing that the direction of systematic search is promising and can potentially shed light on several open challenges in the field, such as generating multi-edit patches and improving patch quality. For example, our approach can fix 39 more bugs and generate thousands more high-quality patches when compared to the baseline GenProg.
- A simple patch ranking mechanism that can effectively rank high-quality patches, for example, to top 10 for 23 out of 24 cases.

## 5.2 Motivating Example

In this section, we use an example to discuss limitations of existing work and benefits of our approach. In Figure 5.1, we show a buggy program taken from the IntroClassJava dataset. The specification of the program is to output the smallest number among the four integer inputs (i.e., a, b, c, and d). Due to incorrect usage of relational operators, this buggy program would fail if the smallest number appears more than once in the inputs.

With the highlighting in Figure 5.1, we show a patch generated by our approach. The patch contains 3 edits, 2 of which modify operators in Boolean expressions and the last one inserts a statement that is taken from the existing code. Intuitively, Edit 3 makes c the default output,

```
1  if (a < b && a < c && a < d) {
2      smallest = a;
3  } else if (b 1 < ⇒ <= a && b < c && b < d) {
4      smallest = b;
5  } else if (c < a && c < b && c < d) {
6      smallest = c;
7  } else {
8      if (d < a && d < b 2 && ⇒ || d < c) {
9        smallest = d;
10     }
11     3 smallest = c;
12 }
```

Figure 5.1: Motivating example modeled after `smallest-1b31fa5c-003`. Code is simplified for readability.

Edit 1 handles cases where a and b are equal and smallest, and finally Edit 2 checks for cases where d is the smallest. Existing work distinguishes patches that are *plausible* versus *correct*, with the former defined as a patch that passes all given tests and the latter a patch that can meet the specification of the program. The patch shown in Figure 5.1 is a correct patch. It is important to note that there could be different ways of fixing the bug, such as by changing all < operators to <=. In the following, we discuss what makes this bug challenging for existing approaches to fix.

## Edit Expressiveness vs. Search Effectiveness

As motivated in the introduction, the success of existing program repair techniques largely depends on the balance between edit expressiveness and search effectiveness. Edit expressiveness determines what fixing ingredients are available in the search space, and search effectiveness determines how likely a (correct) patch can be found. Existing approaches make different tradeoffs.

**Heuristics-Based.** Heuristics-based approaches have the advantage that different edits can be experimented with easily. Search spaces of different sizes and shapes have been explored to increase the likelihood of capturing correct patches in the search space. For example, GenProg [90, 162] can add/replace/delete statements in the original program, such as Edit 3 in Figure 5.1 where a statement is added. CapGen [163] further enlarges the search space by taking fixing ingredients from expressions in the original program. JMutRepair [103] uses classic mutation operators to tweak existing expressions (e.g., Edits 1–2 in Figure 5.1). Another trend is to specialize the shape of the search space to optimize for certain classes of faults. PAR [71] uses specific fixing templates that are mined from human patches. SPR [95] and Prophet [97] generate specific edits for common mistakes such as adding conditional checks and memory initialization statements. Overall, the community is moving toward increasingly larger search spaces of edits.

The larger the space of possible fixing ingredients, the more challenging the search for a patch [96]. Even just executing tests for every single-edit patch can take a long time, and when exploring combinations of fixing ingredients the search space explodes such that executing tests for all possible patches is no longer feasible. Instead, existing approaches largely rely on different

forms of heuristic search. GenProg [90, 162] uses genetic programming, with the assumption that partial patches can be evolved into correct ones, but a weak and non-monotone fitness function can potentially discard partial patches that fail more tests. RSRepair [133] uses random search. AE [161] uses a deterministic heuristic traversal of the single-edit space. SPR [95] uses carefully designed stages to prioritize certain classes of edits. Prophet [97] and CapGen [163] rely on statistical models learned from human patches to prioritize more likely edits. Despite the recent effort of improving search strategies, the search space has not been *systematically* explored. Existing approaches can easily generate the 3 fixing ingredients in Figure 5.1, but cherry picking these three fixing ingredients out of a potentially large pool remains challenging. A recent empirical study suggests that existing search strategies suffer from the increasingly larger search spaces, generating fewer correct patches due to timeout or stopping when finding plausible but incorrect patches [96]. In fact, our approach identified 10177 plausible patches for the bug shown in Figure 5.1, among them only 3 are correct. Cherry picking the 0.03% of correct patches from a much larger pool of plausible patches remains challenging without a systematic search.

**Semantics-Based.** Semantics-based approaches can effectively explore search spaces by delegating the search to program synthesis techniques, but they often restrict themselves to small edits *at the expression level* (e.g., tweaking Boolean expressions in `if` conditions) to make the scope of program synthesis tractable. Edit expressiveness is mainly restricted by the scalability of constraint solvers and the types of theory they can reason about [86]. Thus, existing work is unlikely to generate the patch shown in Figure 5.1 because inserting a statement remains challenging for program synthesis. Although in theory one can synthesize a patch that only changes boolean expressions for the specific bug in Figure 5.1 (e.g., changing all < operators to <=), whether or not such a multi-edit patch can be synthesized remains an open question because multiple expressions across different `if` conditions need to be synthesized.

## Open Challenges

Several open challenges of automatic program repair boil down to the main challenge of maintaining a good balance between edit expressiveness and search effectiveness, which we address in this work by performing a *systematic search* using variational execution. In the following, we discuss a few open challenges our approach can potentially shed light on:

- **General-Purpose Repair.** Recent semantics-based approaches can also perform a *systematic search* by using symbolic execution and program synthesis, but at the cost of using small expression-level edits [34, 86, 107]. Our approach does not make any assumptions about the types of edits, so we can potentially generate more general-purpose patches for more diverse types of bugs than existing semantics-based approaches. Note however that, the choice of edit templates influences the size and shape of the search space. The edit templates are a finite set, but our approach does not explore all *possible* insertions of code at all locations. Our work performs a systematic search, but with regard to a *finite* set of concrete edits encoded in a meta-program (Section 5.4), whereas semantics-based approaches can often traversal a potentially *infinite* space of simple edits by using symbolic

execution and program synthesis.

- **Multi-Edit Repair.** Recent empirical studies suggest that making multiple edits is common when developers fix bugs [61, 175], but automatically generating multi-edit patches remain challenging. In theory, multi-edit patches are under the radar of heuristics-based approaches such as GenProg, but oftentimes the search space is so large that finding them based on heuristics takes time. With an efficient systematic search, multi-edit patches are guaranteed to be found if they exist in the given search space.
- **Patch Quality.** Long and Rinard [96] in a recent study discovered that plausible patches outnumber correct patches by far, making it difficult to prioritize searching correct patches. Our work contrasts most existing approaches for improving patch quality, which typically *restrict* the search space by reducing the edits considered. The work of Tan et al. [150] is the most extreme example where highly specific patterns are used to exclude likely low-quality patches, and this work implicitly informs most other modern techniques in the edit templates they consider. In contrast, our approach diverges in a fundamental and philosophic way from most existing work, in that we do not attempt to restrict the types of edits considered, but rather include more diverse edits to enrich the search space. With an efficient systematic search, correct patches are guaranteed to be found if they exist in the search space. Long and Rinard [96] also suggest that repair techniques should leverage information other than the test suite to pinpoint correct patches. Performing systematic search can thus be a viable avenue to obtain a more complete picture of the space that is otherwise infeasible with a heuristics-based search.

There are different ways to achieve a systematic search. Running each possible edit in a brute force fashion is one way, but only works for tiny search spaces. Using program synthesis is much faster than brute force, but with limited types of edits. In this work, we explore variational execution.

## 5.3 Approach Overview

Given a faulty program and a test suite that can expose the fault, our approach automatically generates patches that can pass all provided tests and subsequently ranks all plausible patches based on their runtime behavior. Reminiscent of how we use variational execution in mutation testing (Section 4.3), our approach consists of 4 steps, as illustrated in Figure 5.2.

1. Similar to most program repair approaches, we use a standard fault localization technique to find suspicious locations in the buggy program. Using the fault localization information, we then generate a set of different edits by applying a predefined set of edit templates to suspicious locations.
2. In order to explore all edits and their combinations at the same time, we merge the entire set of different edits into one meta-program by using `if` conditions to encode different edits. Each `if` condition is guarded by a Boolean variable (e.g., `E1, E2` in Figure 5.2) that controls whether the corresponding edit should be included in a patch (Section 5.4).
3. We run the meta-program with variational execution against a given test suite. In contrast to normal test execution that outputs which tests fail, variational execution reports the combinations of edits that pass the entire test suite. To compactly encode combinations of

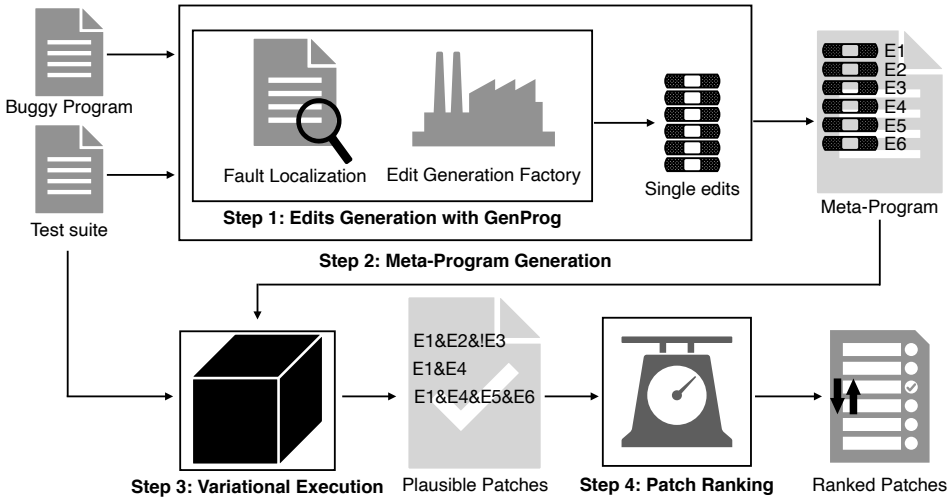



Figure 5.2: Overview of our approach. Steps are marked and surrounded with boxes.

edits, variational execution represents them as propositional formulas, such as $E1 \wedge E2 \wedge \neg E3$ in Figure 5.2, which represents patches that include Edit 1 and 2 but not Edit 3 (Section 5.5).

4. To distinguish high-quality patches from low quality ones, we use the dynamic information obtained from variational execution to rank patches, e.g., prioritizing patches that change less program state, because variational execution can often output many plausible patches, but not necessarily ones that generalize beyond the given test suite (Section 5.6).

## 5.4 Meta-Program Generation

We combine a large set of edits in a single meta-program. The edits in this program, and their interactions, form the search space within which we later search for patches.

We use *genprog4java*[1] to generate the set of mutants that form the search space. We decided to build our approach on top of GenProg because of its conceptual simplicity. GenProg first uses a fault localization technique to rank likely statements based on how many times statements are executed by passing and failing tests, favoring statements that are mostly executed by the failing tests (other fault localization approaches could be plugged in as well, but are generally orthogonal to our main contributions). In those likely faulty statements, GenProg then applies edit templates to generate single edits. Each edit is generated by applying one edit template once.

We inherit three classic edit templates from GenProg: `APPEND`, `REPLACE`, and `DELETE`. They append, replace, and delete code at the statement level of AST. The candidate statements for appending and replacing are taken from the original buggy files, based on the assumption that fixing ingredients are close, known as the plastic surgery hypothesis [11]. To enrich the search space, we add 5 generic expression-level mutation operators to GenProg, based on evidence that shows their effectiveness [121, 122]: Arithmetic Operator Replacement (`AOR`), Relational

[1] https://github.com/squaresLab/genprog4java

```
1  if (a < b && a < c && a < d) {
2      smallest = a;
3  } else if (b 1 (e1 ?  b <= a :  b < a) a && b < c && b < d) {
4      smallest = b;
5  } else if (c < a && c < b && c < d) {
6      smallest = c;
7  } else if (2 e2 ?  (d < a && d < b || d < c) :  (d < a && d < b && d < c)) {
8      smallest = d;
9  } else {
10 3   if (e3)
11 3     smallest = c;
12 3   else
13 3     smallest = Integer.MAX_VALUE;
14 }
```

| Test | Original | Passing Condition |
|---|---|---|
| `assert smallest(1, 2, 3, 4) == 1` | ✓ | `true` |
| `assert smallest(1, 1, 1, 1) == 1` | ✗ | `e3` |
| `assert smallest(1, 1, 2, 3) == 1` | ✗ | `e1` |
| `assert smallest(1, 2, 3, 1) == 1` | ✗ | `e2` |
| Whole test suite | ✗ | `e1` ∧ `e2` ∧ `e3` |

Figure 5.3: The upper part is an example of meta-program that encodes 3 edits for `smallest-90834803-005`. The lower part is a manually constructed test suite for demonstrating how variational execution is used. The Original column reports test outcomes of the original buggy program.

Operator Replacement (ROR), Logical Connector Replacement (LCR), Absolute Value Insertion (ABS), and Unary Operator Insertion (UOI). Other edit templates (e.g., recently developed ones that target specific fault classes [96, 139, 163]) can be easily integrated as extensions. We modify GenProg to repetitively mutate the given buggy program until a specified number of different edits are generated. After filtering out edits that are not compilable (standard GenProg step), we merge the remaining edits into one meta-program.

Similar to meta-program generation in Chapter 4, we encode all edits as optional code paths (guarded by `if-then-else` statements and expressions) into a single meta-program. For each edit, we introduce a Boolean option (e.g., global static field in Java) that decides whether the original or the edited code is executed, as illustrated in Figure 5.3. Our encoding of edits is flexible, allowing other more recent types of edits to be expressed freely in the meta-program. This way, our approach maintains the expressive power of existing heuristics-based approaches.

The idea of generating a meta-program for program repair is not new, but rather an important preprocessing step that defines a search space for variational execution to search (cf. Chapter 4). The repair techniques of Weimer et al. [161] and Kern and Esparza [68] also generate conceptually similar meta-programs. Our work on finding higher-order mutants (Chapter 4) and Devroey et al. [30] used similar encodings for speeding up mutation analysis. However, we implemented our meta-program generation from scratch on top of *genprog4java* because the one we used in Chapter 4 for mutation testing does not perform fault localization or produce statement-level changes.

Edit generation as part of creating the meta-program involves randomness, i.e., selecting

edits from a potentially very large pool of possible edits (`APPEND` alone can produce many edits in most programs). We use the *undeterministic* weighing scheme from GenProg to generate edits and repeatedly invoke GenProg until we generate a predefined number of distinct edits. Due to randomness, a different run might yield a different search space, but we can alleviate the randomness by generating a large pool of edits, which we will show in Section 5.8.

## 5.5 Systematic Search with Variational Execution

Given a meta-program and its test suite, we use variational execution to determine what combinations of edits can pass all tests. We can achieve high search effectiveness because we can explore the search space in an *efficient* and *comprehensive* way, as opposed to heuristics-based approaches that validate one patch candidate at a time (hence *inefficient*) and rarely systematically explore combinations of edits due to a lack of reasonable measure for partial patch (hence *incomprehensive*).

Similar to other applications of variational execution discussed in previous chapters, the key insight in using variational execution is that most test executions are similar or even identical when edits modify the buggy program. Many edits have only minor and local effects on control-flow and program state of some test executions, whereas most computations are the same or similar independent of whether an edit is applied. Variational execution exploits those sharing opportunities among many similar executions with minor differences, regarding both data flow and control flow. It shares the same value for a variable across all or many possible combinations. More technical details of variational execution are available in Chapter 3. We use VarexC, but make several adjustments to better fit the characteristics of this search space.

### Finding Plausible Patches with Variational Execution

To identify which combinations of edits can pass all tests and fix the buggy program, we simply execute each test for the meta-program (with symbolic values representing all edits) and capture for each test the condition under which the test passes. We call this condition a *passing condition*, which is a propositional formula over the symbolic values to compactly capture all combinations of edits that pass the test. Any solution to the propositional formula represents a plausible patch. To get plausible patches that pass all the tests, we simply use a SAT solver or BDD to enumerate all solutions that satisfy the conjunction of all passing conditions.

As of recap of how we use variational execution (cf. Chapter 4), we execute every test in the test suite, setting all variables representing edits in the meta-program as symbolic inputs (e.g., `e1`, `e2`, `e3` in Figure 5.3). When the test execution finally hits an assert statement, we can simply capture the path condition to identify under which conditions the test passes. For example, in Figure 5.3, `assert smallest(1, 2, 3, 4) == 1` passes under condition `true`, meaning that the test can pass with or without any edits. However, `assert smallest(1, 1, 1, 1) == 1` passes under condition `e3`, meaning that it can pass only when Edit 3 is applied, independent of the other edits. Finally, the passing condition for the whole test suite is $e1 \wedge e2 \wedge e3$, suggesting that we need all three edits to fix the program.

We use variational execution to execute test cases one at a time, but prioritizing test cases

that the original buggy program fails. After each test execution, we check if there exist any combinations that can pass the tests executed so far, and continue to the next test only if there remain solutions in the search space. In the scenario shown in Figure 5.3, one of the 3 failing tests will be executed first, and let us suppose the first test is `assert smallest(1, 1, 2, 3) == 1`. If we did not have Edit 1 in the meta-program shown in Figure 5.3, we could terminate our search already after executing the first test, because no combination of available edits would pass this test; there are simply not enough fixing ingredients in the search space. This way, we can narrow down the search space quickly or conclude that no patch exists early on in the searching process.

This way of using variational execution to search for repair is both *sound* in that each repair emitted by our approach passes all given tests, and *complete* with regard to the given search space (i.e., repair must be found if exists within the search space).

## Bounded Search

While variational execution can conceptually explore all possible interactions within the search space, doing so may be expensive when many edits (i.e., fixing ingredients instantiated from edit templates) interact. In addition, patches that combine more than a few edits may be too complex for human developers to be interesting. Hence it can be beneficial to bound the search space.

We extended VarexC to (optionally) bound the search space to limit the number of individual edits in a patch. That is, if we set a bound to be $n$, all patches that comprise more than $n$ edits will be out of consideration. For example, the patch shown in Figure 5.3 is only discoverable if the bound is greater than or equal to 3. Technically, when representing alternative values in variational execution, we prune those alternatives that depend on more than $n$ activated edits; we also do not execute path that have a path condition requiring more than $n$ activated edits.

Note that *within the bound*, our search with variational execution is still *complete* in that part of the search space.

## Partitioning Mode

Orthogonal to bounded search, an alternative way of bounding the search space is to limit the number of individual fixing ingredients, especially for large search spaces that contain thousands of edits.

We extended VarexC to optionally partition the search space, considering at most $n$ edits throughout the search where $n$ is configurable. Different strategies can be used to pick the $n$ edits among all edits in the large search space, such as picking edits based on fault localization suspiciousness or picking edits that have close proximity. As the first attempt, we *randomly* pick $n$ edits *once* at the beginning of variational execution and leave other alternative ways of picking edits to future work.

Similarly, *within the bound*, our search with variational execution is complete in that partition of the search space, but different partitions can be taken to incrementally explore larger portions of the overall search space.

### Fast Mode

In addition to bounding, we implemented another optional optimization for prioritizing test executions that do not throw any exceptions. Many edits introduce behavior in the program that ends in exceptions (e.g., null pointer exceptions, division by zero). As discussed in Section 3.5, we can reliably handle *all* exceptions at the cost of potentially expensive re-execution of variational execution. Although it is possible that those exceptions can be caught in other parts of the program, we have not observed any in practical repair problems. Ignoring conditional exception handling paths can significantly prune the search space and speed up variational execution. This optimization is similar to how S3 uses symbolic execution to run *failure-free* execution paths [86].

In an optional mode we call *fast mode*, we reduce the search space whenever we encounter an exception in a conditional code path (i.e., with a path condition that is not `true`). We simply terminate that code path early and do not join it with the other paths. We record the path condition of the exception in case we want to explore those paths later, which we only do if we do not find any plausible patches without handling exceptions. For example, if a test case expects certain exception to be thrown in order to pass, variational execution will not find any plausible patches with fast mode, but will fall back to normal mode to explore exception-throwing paths. Note that exceptions thrown in all executions are still executed as usual (i.e., not caused by edits, commonly in early parts of the execution trace before reaching the edits).

Our fast mode may introduce incompleteness, because it will prioritize finding patches that do not require catching exceptions thrown by edited code even when exception handling is part of the normal behavior, but we trade that incompleteness with a performance improvement of exploring a smaller search space. Soundness is not affected, identified patches are still passing all tests.

## 5.6 Patch Ranking

With variational execution, we can systematically explore the search space to find many plausible patches if they exist. For example, as we will show in Section 5.8, our approach can find up to 33017 plausible patches for a buggy program. However, recent studies have revealed that oftentimes plausible patches have low quality, to the extent that they might impair developer productivity [151]. It is thus important to distinguish high-quality patches from low-quality ones.

Note that, existing approaches often have an internal component that ranks *patch candidates* to decided in which order to validate them. Our problem of ranking *plausible patches* is inherently different from ranking *patch candidates* in that all plausible patches can pass the provided test suite, so measures such as ranking based on the number of tests passed [90, 162] do not apply to our ranking problem. As suggested by Long and Rinard [96], more information is needed to generate high-quality patches. In this work, we leverage the dynamic information of test executions. We take a two-step approach to ranking plausible patches.

## Minimization

First, we apply minimization to all generated plausible patches. Given a plausible patch composed of multiple edits, minimization removes all edits that are not necessary for passing all tests. Minimization is important for generating concise patches. Patches that can be minimized are often constructed by adding an extra edit on top of smaller plausible patches. The extra edit is typically a code change that does not affect executions of the provided tests. Edits like this may fix things not tested by the test suite or they may introduce new problems not detected by the test suite; or they may add dead code. However, from the perspective of the test suite, they are entirely undistinguishable. There are typically too many plausible patches that could be minimized but we have no magic oracle to verify them automatically (held-out tests are only for academic evaluations and would not be used in a practical setting). To generate concise patches and reduce noise in patch ranking, our approach performs minimization on all plausible patches and only outputs minimized ones.

There are different ways for minimizing patches, such as repeatedly running tests to verify removal of edits one by one, or in a more efficient fashion similar to delta debugging as performed in GenProg [162]. However, in our approach, if a non-minimized patch is found, it is guaranteed that the corresponding minimized one can also be found because of the systematic search enabled by variational execution. Hence, we simply examine the composing edits of plausible patches and only keep those patches that cannot be subsumed by other patches. More formally, given a set of plausible patches $\mathcal{P}$, where each patch $p \in \mathcal{P}$ consists of a set of edits $e \in p$, a patch $p$ is minimal in $\mathcal{P}$ iff there is no other patch with a subset of these edits that also passes the test suite:

$$p \text{ minimal in } \mathcal{P} \Leftrightarrow \nexists p' \{p' \in \mathcal{P} \wedge p' \subset p\}$$

## Patch Ranking Heuristics

Given a set of minimized plausible patches, we use a heuristic-based ranking strategy to prioritize patches that are more likely to have high quality. Our heuristic leverages dynamic information about runtime behaviors of patches, such as what variables are modified, what branches are taken, and what lines are executed. It has been shown in a recent study that dynamic information is useful for distinguishing *correct patches* and *plausible but incorrect patches* [168]. This kind of dynamic information can often be obtained by recording and aligning execution traces, a process that can be expensive when performed in large scale [109]. Since variational execution can aggressively share common execution paths, we can take this unique opportunity to use it as an on-the-fly alignment tool to record additional runtime information while running meta-programs against tests, similar to how it has been used to record runtime data for debugging [109].

In a recent study, Xiong et al. [168] demonstrate how dynamic information can be used to classify if a plausible patch is correct or not, by measuring the similarity of execution traces between the patched program and the original program. The key idea is that, a correct patch should have little effect on passing tests (hence high similarity between execution traces), but should cause failing tests to behave differently (hence certain dissimilarity). Although technically Xiong et al. [168] address a classification problem rather than a ranking problem, we suspect

that the underlying principles are transferable. We adopted their central ideas and made some adaptations to their formulas to compute a score for each plausible patch. We also extended their ideas by exploring other types of dynamic information. Note that our patch ranking mechanism is independent of variational execution. Other repair approaches can use external tools to obtain necessary runtime information for ranking, similar to Xiong et al. [168].

For each plausible patch, we compute a penalty based on the distance between execution traces of the patched program and the original program. Xiong et al. [168] define a distance by computing the longest common subsequence (LCS) of executed statements. We simulate their distance by using executed lines, and define $distance_{line}$ as follows, where $t_p$ and $t_o$ represents the trace (as executed lines) of the patched and original program, respectively.

$$distance_{line}(t_p, t_o) = 1 - \frac{|LCS(t_p, t_o)|}{max(|t_p|, |t_o|)}$$

While executed lines capture control flow information at a fine-grain level, we can also compute a coarse-grain control-flow distance by counting how many control flow branches were taken differently by the patched program. Hence, we define $distance_{if}$ as follows, where $\Delta_{if}$ computes the number of different control-flow branches and $\bigcup_{if}$ computes the total number of unique control-flow decisions.

$$distance_{if}(t_p, t_o) = \frac{\Delta_{if}(t_p, t_o)}{\bigcup_{if}(t_p, t_o)}$$

Finally, we use data-flow information to define a distance based on changes to program state. We define $distance_{var}$ as follows, where $\Delta_{var}$ computes the number of variables and fields that are modified differently at any point in the execution, and $\bigcup_{var}$ computes the total number of unique variables and fields.

$$distance_{var}(t_p, t_o) = \frac{\Delta_{var}(t_p, t_o)}{\bigcup_{var}(t_p, t_o)}$$

Given these types of distance, we compute a penalty for each plausible patch as shown in the formulas below, where $p$ and $o$ represent the patched program and the original program, $\tau$ represents a test case, and $\mathcal{T}$ represents all the tests. Different types of distance can be plugged in to rank patches based on different dynamic information. Inspired by Xiong et al. [168], we treat positive tests and negative tests differently: For passing tests, we take the maximum distance among all passing tests, based on the intuition that passing tests should behave similarly to the original buggy program. Taking the maximum can help us penalize the patch for the most abnormal execution. For failing tests, we take the average because the patched program is expected to behave differently, but how differently can vary across tests. Finally, we sum up the penalties from positive tests and negative tests to compute a final penalty. All plausible patches are ranked in ascending order of penalty.

$$pos(p, o) = max(\{distance^{\tau}(t_p, t_o) \mid \tau \in \mathcal{T}\})$$
$$neg(p, o) = avg(\{distance^{\tau}(t_p, t_o) \mid \tau \in \mathcal{T}\})$$
$$penalty(p, o) = neg(p, o) + pos(p, o)$$

## 5.7 Implementation

We built our meta-program generator on top of *genprog4java*, reusing its fault localization and edit generation infrastructure. First, we modified *genprog4java* to mutate the buggy program repeatedly until we get a large pool of distinct edits. Next, we filter out edits that could make the program fail to compile. We then apply all the remaining edits to the original program by wrapping new edits in `if` conditionals, nesting multiple conditionals if necessary to apply multiple edits at the same location. Finally, we run the generated meta-program against the original test suite to make sure behaviors are the same as the original program when all edits in the meta-program are disabled.

Similar to generating meta-programs for finding higher-order mutants (Chapter 4), putting too many edits into a single method can result in methods that are too large for the JVM or variational execution. Whenever possible, we push edits into small methods, as discussed in Chapter 4 for expression-level changes. For statement-level changes, we automatically extract relevant statements into small methods after refactoring, such as replacing local variables with fields. Details can be found in our implementation.

**Limitations and workaround.** Due to the limitations of VarexC as a research prototype (Section 3.5), we had to perform some minor refactoring in our subject systems, for example replacing a collections library by a different one. Fast mode is also particularly effective in the current implementation of VarexC because it avoids an unusually large performance overhead of exception handling. We also failed to execute some tests with variational execution in some subject systems; in those cases, we derive potential patch candidates from those test cases we can execute and then subsequently execute other tests individually for those patch candidates to see whether they actually pass all provided tests. We use variational execution to execute all failing tests at the very least, in order to narrow down the search space as much as possible. This way, variational execution is still beneficial to identify a usually fairly small set of patch candidates for this further analysis.

As in Chapter 4, we observed *combinatorial explosion* in a few cases: many random edits cause a single variable to have more than 8000 alternative concrete values. Moreover, edits in automatic program repair tend to create more infinite loops and endless recursive calls than higher-order mutation testing, likely because statement-level changes cause these issues more easily than expression-level changes. For this reason, we carefully bound the number of executed basic blocks and the height of the method call stack, terminating execution for the parts of search space where a threshold is reached. This limitation might affect completeness of our approach if the threshold is not set generously enough, but our approach remains sound. We argue that

this is the essential complexity of the mutated program and it would be equally challenging for other search strategies to systematically explore such a complex search space.

## 5.8 Evaluation

We evaluate our approach in multiple dimensions such as search effectiveness and patch quality. Where available, we compare our results with the state-of-the-art approaches by taking numbers verbatim from prior work, because reproducing previous results can be expensive and time-consuming [33]. Our experiment setup and evaluation data are publicly available online.

### 5.8.1 Research Questions

We ask the following research questions:

| | |
|---|---|
| RQ1 (Effectiveness): | How effective and efficient our approach is in finding patches within a large search space of fixing ingredients? |
| RQ2 (Patch Quality): | To what extent do our generated patches overfit to the provided tests? |
| RQ3 (Fixing Ingredients): | To what extent can our approach make use of different kinds of fixing ingredients? |
| RQ4 (Multi-Edit): | How effective is our approach in generating multi-edit patches? |
| RQ5 (Patch Ranking): | How effective is our patch ranking? |

Central to an automatic program repair tool is its ability to identify patches. **RQ1 (Effectiveness)** as an overarching research question investigates the overall effectiveness and efficiency of our approach. We measure effectiveness in terms of the ability to find patches and efficiency in terms of the time it takes to find the first patch. Next, we examine the quality of our identified patches by answering **RQ2 (Patch Quality)**. It has been discovered that test-based program repair often generates patches that fail to generalize to other tests beyond the ones used for repair [86, 113, 145]. In theory, a systematic search like ours would increase the chance of identifying high-quality patches if they exist in the search space. In the next two research questions, we further investigate factors that make our approach effective. While it is important to include the right fixing ingredients into the search space, it is also critical to use and gather all necessary ingredients to form a patch. In **RQ3 (Fixing Ingredients)**, we investigate how well the systematic search of variational execution can make use of the fixing ingredients in the search space. **RQ4 (Multi-Edit)** concerns multi-edit patches, which remains one of the open challenges of automatic program repair [91, 113]. Multi-edit patches are difficult to find in a large search space because there could be exponentially many ways of combining fixing ingredients. We hypothesize that a systematic search like variational execution can increase the likelihood of gathering multiple edits to form a patch. Finally, as we will see in our results, there are usually abundant plausible patches in the search space, which is also pointed out by prior work [96]. It is thus important to distinguish high-quality patches from the rest of the plausible patches that likely fail to generalize. **RQ5 (Patch Ranking)** investigates whether our patch ranking is effective at isolating high-quality patches from low-quality patches.

### 5.8.2 Datasets

We use two existing datasets for our evaluation, both of which are commonly used to evaluate program repair approaches. In Table 5.1, we show basic statistics of the subjects.

The IntroClassJava dataset [35] is ported from a set of small C programs written by undergraduate students when taking an introductory programming course [145]. Each buggy program is a homework submission for a basic programming task. Since programs in the IntroClassJava dataset are written by different students, we report the average lines of code (LOC) for each subject in Table 4.1. These programs are small, but there are several advantages to using them in our evaluation. First, prior work shows that these bugs feature potentially complicated fixes despite their small program size [33, 86]. Second, these small programs yield a reasonably sized search space, providing a perfect ground for in-depth comparison among different search strategies. Third, all of these programs have their errors in one method so the impact of fault localization is minimized, allowing us to focus on comparing search strategies. Finally, the IntroClassJava dataset was originally proposed to evaluate patch quality because the specifications of these introductory programming tasks are simple and each subject comes with two distinct test suites. Both test suites are comprehensive in terms of coverage so they are useful for evaluating generality of patches [145]. In fact, we observe that these simple programs barely use advanced language features beyond simple integer computations and if statements, making it tractable to formally reason about patch correctness using our customized symbolic execution engine, as discussed in Section 4.6. We excluded some IntroClassJava bugs that do not pass any provided tests, in which case the fault localization component we inherit from GenProg would fail.

The Defects4J [61] dataset is a carefully curated corpus of bugs taken from popular open-source projects. Existing program repair work often uses a subset of this dataset to demonstrate the usefulness of fixing real bugs in practice [23, 58, 139, 167]. For our evaluation, we focus on two subjects that (1) have been commonly used in previous work; (2) have the largest number of bug counts; and (3) have the largest code base in terms of lines of code (LOC). Our approach is conceptually generalizable to other subjects, but we focus on Math and Closure to maximize the number of comparable results and minimize the engineering effort needed to set up variational execution, specifically for handling environment barrier as discussed in Section 3.5, which is likely common to other heavyweight semantics-based approaches. In Table 5.1, we report the number of bugs we actually used. Some bugs from Math and Closure were discarded because of engineering issues discussed in Section 5.7 and Section 3.5.

### 5.8.3 Meta-Program Generation

We generate one meta-program for each buggy program and use it consistently in all research questions, because generating meta-programs can be expensive (e.g., up to 8 hours for one bug). We use all 8 edit templates discussed in Section 5.4 but assign a lower weight to `UOI` and `ABS` (0.1 as opposed to 1.0 for others) because they can easily create lots of edits that are of little value. For example, both `UOI` and `ABS` can be applied to any numeric constants, variables, and fields, potentially yielding many patch candidates that are semantically equivalent to the original program. For IntroClassJava we use GenProg to create up to 500 different edits, compile

Table 5.1: Evaluation Subjects

| Subject | Description | LOC | Test LOC | #Bugs/#Total | #Tests |
|---|---|---|---|---|---|
| `median` | median of 3 integers | 78 | 130 | 46/57 | 13 |
| `smallest` | smallest of 4 integers | 80 | 158 | 43/52 | 16 |
| `digits` | digits of an integer | 83 | 153 | 68/75 | 16 |
| `grade` | numeric grade to letter grade | 82 | 174 | 79/89 | 18 |
| `checksum` | checksum of a string | 89 | 172 | 11/11 | 16 |
| `syllables` | count syllables | 88 | 152 | 12/13 | 16 |
| `Math` | Apache Commons Math library | 85k | 19k | 87/106 | 3,602 |
| `Closure` | Closure compiler | 90k | 83k | 97/176 | 7,927 |

LOC numbers for IntroClassJava are counted using SLOCCount and averaged over all bugs.
LOC numbers for Defects4J are taken from the original work [61].
#Bugs/#Total denote the number of bugs included in our evaluation and the total number of bugs available in the dataset, respectively. Some bugs were discarded due to setup issues.
#Tests denotes the number of test cases.

them individually, and merge compilable ones into one meta-program. The bound of 500 edits is determined based on our experience. For Defects4J we increase the bound to 5000 to account for the bigger program size.

### 5.8.4 RQ1 (Effectiveness)

**Experiment Setup.** To measure the effectiveness of our search strategy, we conduct an experiment in which we apply VarexC on different bugs and measure how many bugs can be fixed. Consistent with prior work, we distinguish bugs that are fixed by *plausible patches* and those fixed by *generalizable patches*. We will define and discuss different kinds of patches in RQ2 (Patch Quality), but intuitively, generalizable patches have higher quality because they can pass tests that are not provided for generating repairs, hence generalizable.

We run VarexC on each pre-computed meta-program and set a time limit of 3 hours for IntroClassJava and 6 hours for Defects4J. Our time limit of 3/6 hours is on par with recent prior work [33, 58, 139]. As discussed in Section 5.5 and Section 5.7, we can configure VarexC in different ways to adjust the search for different types of programs and different shapes of the search space. For example, programs that involve many recursive calls require a bigger maximum stack height than programs without recursive calls. We use the same settings of VarexC across all executions, but manually adjust certain parameters such as *maximum stack height* and *maximum block count* for a few executions if we observe runtime errors. Since expensive loops and recursive calls can sometimes hijack the overall search in VarexC, we adjust the search bound of VarexC by carefully setting *search degree* and *maximum edits*: For IntroClassJava, we set the search bound to degree 3 with maximum 500 edits if the buggy program does not involve any loops. For those that have loops, meta-program generation tends to insert the same loops multiple times due to limited fixing ingredients in the small programs, so we start with a search bound of degree 2, and continue with degree 3 if there is remaining time budget. For Math and Closure, loops and recursions are common, so we increase the search bound gradually, starting from degree 1 with all edits, then degree 2 with a sample of 500 edits, and finally degree 3 with a sample of 300 edits. Multiple samples can be taken until the time

limit is reached.

It is important to note that we exclude the time of meta-program generation from all time measurement in our evaluation because it is a pre-processing step for variational execution. In contrast, heuristics-based approaches tend to generate and validate patch candidates individually on-the-fly, and semantics-based approaches often include the synthesis step when measuring execution time. We configured our baseline GenProg to search over the same meta-programs for a fair comparison with VarFix. But *efficiency* comparison with prior work should be made with caution. Similarly, we report how many bugs prior work can fix, but *effectiveness* should be compared with caution because existing approaches use more tailored fixing ingredients. The novelty of our work lies solely in the search strategy, but our work is orthogonal to recent advances in creating diverse fixing ingredients.

To evaluate *effectiveness* of our approach, we measure how many bugs our approach can fix and compare our results with state-of-the-art approaches [23, 58, 86, 139, 163, 167, 170, 172]. To mitigate the confounding factor of having different fixing ingredients in the search space, we adapt GenProg to use the same meta-program as VarexC while performing genetic search. We choose GenProg for its conceptual simplicity. GenProg remains a reasonable baseline despite almost 10 years of research in program repair [23, 58, 163, 167]. We configure GenProg to run continuously and record all identified patches during the search until timeout. For IntroClassJava, we configure GenProg to run until it has tried 20 different seeds or exceeded a time budget of 3 hours. For Defects4J, we scale up the search budget to 40 seeds or 6 hours of execution. Since edit generation and compilation time are excluded, our search budget for GenProg is generous, allowing it to sufficiently explore the search space using genetic search, in the same spirit of Long and Rinard [96] where a generous budget was given to find as many patches as possible in a given search space.

To evaluate *efficiency* of our approach, we measure the time it takes to generate the *first* plausible patch. Again, we compare VarexC with GenProg using the same meta-programs. GenProg remains a representative heuristics-based repair approach because validation of patch candidates still dominates the overall search. The search heuristics used to rank patch candidates might affect efficiency, especially in cases where validating each patch candidate requires non-trivial amount of test execution. To mitigate the impact of search heuristics, we make the validation in GenProg more efficient by sampling 10% of the provided tests when validating patch candidates. If all the tests in the sample pass, GenProg continues with the rest of the tests. This way, an efficiency comparison with GenProg is likely generalizable to other heuristics-based approaches.

We set up the experiments in Docker containers and ran all performance measurement on Amazon Fargate. Each Fargate instance has 2 vCPU and 16 GB of RAM. Altogether, the experiments for RQ1 took more than 5000 hours of CPU time.

**Threats to Validity.** Before we show results, we discuss potential threats to validity:

- We discarded some Math and Closure bugs because of difficulties of setting them up for variational execution, due to issues like unsupported API calls. As discussed in Section 3.5, other semantics-based approaches that rely on symbolic execution are likely to face similar difficulties, which can often be addressed with more engineering effort such as writing

more model code to abstract those API calls [166]. The exclusion of these bugs might affect our findings to some extent, but we do not expect a systematic bias because the existence and quality of patches are largely independent of specific API calls.

- Direct comparisons of raw numbers with prior work should be made with caution because the setup could be drastically different, such as using different fault localization techniques and different fixing ingredients. To objectively evaluate our approach, we compare with GenProg within a consistent setup.
- Randomness might affect several components of our experiment, such as generating meta-programs, sampling edits for variational execution, and genetic search of GenProg. To limit the impact of randomness, we intentionally generate a large pool of applicable edits when we generate meta-programs. We also take different samples or seeds when applicable.
- As with all existing approaches, our approach is evaluated on a limited number of bugs and thus generality is unclear. Technical limitations in the current implementation may make it more challenging to run other programs, but the conceptual ideas of variational execution as a technique are generalizable. Following prior work, we use the most popular datasets in our evaluation. Recent studies criticize that existing program repair approaches tend to overfit to specific benchmarks such as Defects4J [33], but we argue that the overfitting issues mainly come from using fixing ingredients that are specific to certain programs. The novelty of our approach lies in the effective and efficient search strategy. We pick our eight edit templates because they are basic, general and well understood, but our approach is generalizable to different kinds of fixing ingredients. A further evaluation using more advanced edit templates on other subjects is interesting but out of scope.
- Finally, potential errors in our tooling and manual analysis might affect the results. To mitigate potential errors, we performed rigorous testing on our tooling and took due diligence to verify our results. We present raw data and concrete examples in the paper, and make our tools publicly available for open investigation.

**Results.** For IntroClassJava, as we can see in Table 5.2, our technique significantly outperforms the state of the art on all subjects, fixing more bugs with generalizable patch. The only exception is S3, which can produce generalizable patches for 7 more `smallest` bugs than VarFix. We speculate that S3 performs better on `smallest` because S3's search space includes more diverse fixing ingredients, (e.g., adding or removing Boolean expressions in branch conditions) [86]. Even with the simple fixing ingredients, VarFix can uniquely produce generalizable patches for 40 bugs (e.g., 11 for `median`, 8 for `smallest`, and 19 for `digits`) when compared to the state of the art.

Comparing to GenProg on the same meta-programs, VarFix outperforms GenProg on all subjects, fixing 28 more bugs with generalizable patches. In fact, the bugs fixed by GenProg is a strict subset of those fixed by VarFix, indicating that the search strategy of VarFix is much more effective at identifying patches that are available in the given search space.

For Defects4J, as shown in Table 5.3, we observed similar patterns as IntroClassJava, although the repair problems of Defects4J are much larger in terms of lines of code and number of tests (Table 5.1). While numbers are in the same ballpark, VarFix produces generalizable

Table 5.2: Patch generation for IntroClassJava (Generalizable/Plausible).

| Subject | #Bugs | VarFix | GenProg | CapGen | S3 | Nopol | jMutRepair | ARJA |
|---|---|---|---|---|---|---|---|---|
| `median` | 57 | 23/32 | 13/18 | 8/- | - | 16 | 7 | 7 |
| `smallest` | 52 | 15/38 | 2/18 | 11/- | 22/- | 12 | 9 | 9 |
| `digits` | 75 | 22/30 | 16/26 | 3/- | - | 2 | 4 | 6 |
| `grade` | 89 | 4/4 | 4/4 | 3/- | - | 2 | 4 | 0 |
| `checksum` | 11 | 0/2 | 0/1 | 0/- | - | 0 | 0 | 1 |
| `syllables` | 13 | 0/1 | 0/1 | 0/- | - | 0 | 0 | 0 |
| Total | 297 | 64/107 | 35/68 | 25/- | 22/- | 32 | 24 | 23 |

A hyphen (-) denotes missing data.
#Bugs denotes the total number of bugs for the subject.
For VarFix, GenProg, CapGen, and S3, each cell shows the number of bugs patched by **generalizable**/**plausible** patches.
For Nopol, jMutRepair, and ARJA, each cell shows the number of bugs patched by **plausible patches** since patch quality is not evaluated [33].

Table 5.3: Patch generation for Math and Closure (Generalizable/Plausible).

| Subject | #Bugs | VarFix | GenProg | Hercules | SimFix | ssFix | CapGen | JAID |
|---|---|---|---|---|---|---|---|---|
| Math | 106 | 11/24 | 7/16 | 20/29 | 14/26 | 10/26 | 13/- | 5/8 |
| Closure† | 176 | 6/11 | 0/1 | 8/13 | 5/7 | 2/11 | -/- | 7/10 |
| Total | 282 | 17/35 | 7/17 | 28/42 | 19/33 | 12/37 | 13/- | 12/18 |

Each cell shows the number of bugs patched by **generalizable**/**plausible** patches.
#Bugs denotes the total number of bugs for the subject.
Numbers for existing approaches are taken from the corresponding papers[23, 58, 139, 163, 167].
CapGen was not evaluated on Closure and the paper only reports correct patches.
†: Closure-62 and Closure-63 are the same. We count only one of them and manually adjust numbers of other approaches for consistency.

patches for fewer bugs than the state of the art except ssFix on either Math or Closure. Again, we speculate that these approaches fix more bugs with generalizable patches because they use more diverse fixing ingredients. For example, Hercules can fix `Math-24` by inserting a method call that is absent from the buggy source file. A fixing ingredient like this is beyond the scope of our search space because we only reuse code from the source file under repair. That said, even with our simple fixing ingredients, VarFix can uniquely fix 7 bugs with generalizable patches (4 for Math and 3 for Closure). We suspect that VarFix can repair bugs that existing approaches did not fix because our search space is more expressive (i.e., we do not restrict the types of edits) and our research is systematic with regard to the given search space (i.e., we do not use heuristics to traverse the space).

When compared with GenProg, VarFix again shows clear advantage on all subjects. For Closure, the results of VarFix are noticeably better than GenProg. We observed that Closure bugs tend to involve thousands of test cases, making repeated validation of patch candidates very expensive for GenProg. For this reason, GenProg can only explore limited number of plausible patches and thus search effectiveness is impaired. In contrast, VarFix only executes test cases once by using variational execution to aggressively share test executions of all plausible patches. These results suggest that the search strategy of VarFix remains effective at identifying patches in large search spaces of large programs with thousands of tests.

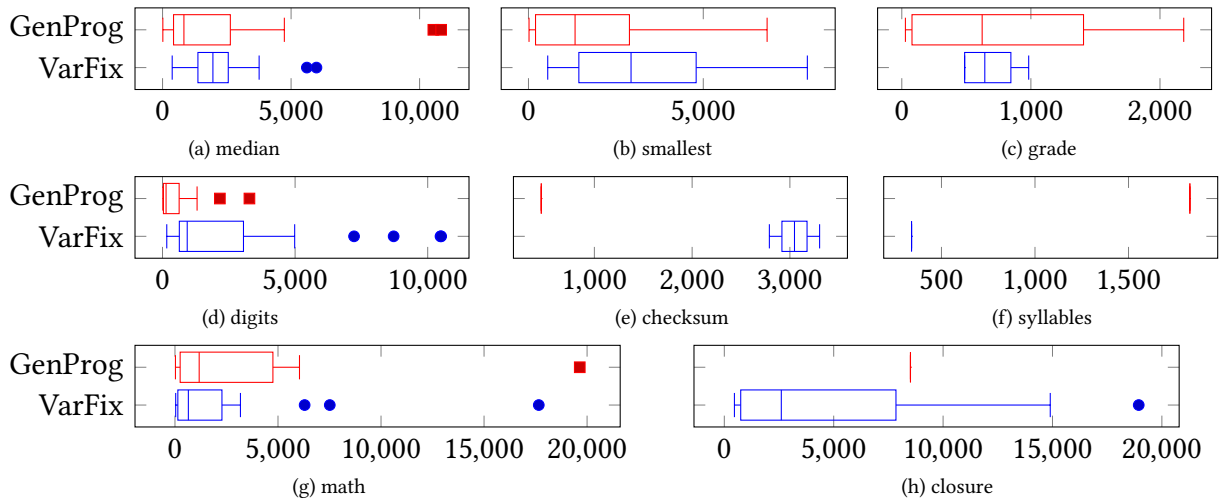



Figure 5.4: Efficiency comparison between VarFix and GenProg. In each box plot, we show the time taken to find the first patch for all the bugs of the given subject. The horizontal axis displays time in seconds.

In Figure 5.4, we compare search *efficiency* in terms of *time to find the first plausible patch*. For IntroClassJava, VarFix tends to take slightly more time to find the first patch, mainly because variational execution explores all the patch candidates at the same time. But when variational execution terminates, VarFix can usually find many more patches than GenProg (not shown in Figure 5.4 but reflected in Table 5.2 and Table 5.3). For Defects4J, VarFix consistently outperforms GenProg because of two reasons. First, the overhead of variational execution is offset by larger search spaces and more expensive test executions when fixing bugs in large programs. Second, we configured VarFix to gradually increase the search bound to prevent variational execution from getting stuck at expensive loops or recursive calls. That is, VarFix attempts to fix the given bug with only 1 edit, then 2 edits, and finally 3 edits. If a single edit is sufficient to fix the bug, which is the case for many bugs in Defects4J [61], VarexC can generate patches quickly. Similarly, we could configure VarFix to gradually increase the search bound on the IntroClassJava dataset if our goal is to optimize for search efficiency. However, we argue that the overall repair time for IntroClassJava is reasonable and the differences between VarFix and GenProg are small.

**Summary**: RQ1 (Effectiveness)

How effective and efficient our approach is in finding patches within a large search space of fixing ingredients?

- A direct comparison with GenProg using the same meta-programs reveals that VarFix is strictly more effective in identifying generalizable patches, fixing 29 more IntroClassJava bugs and 10 more Defects4J bugs with generalizable patches.
- VarFix significantly outperforms prior work on IntroClassJava and can uniquely fix 7 Defects4J bugs. Although VarFix repaired fewer Defects4J bugs overall, the main reason, we speculate, is that prior work uses more tailored fixing ingredients.
- The efficiency of VarFix is comparable to lightweight approaches like GenProg. The overhead of VarFix becomes less obvious for larger programs with more tests.

### 5.8.5 RQ2 (Patch Quality)

**Experiment Setup.** To study patch quality, we use automated analyses and manual checking to categorize patches generated by VarFix into different groups. We distinguish high-quality patches that are generalizable beyond the tests used for repair from patches that overfit to the provided tests. Existing work has taken different measures to assess patch quality and used different terminologies to categorize patches. In the following, we describe the terms we use and the measures we take to identify them:

- **Plausible patch** is a patch that can pass all the provided tests. All the patches identified by VarFix, GenProg or other existing tools are plausible patches.
- **Generalizable patch** is a plausible patch that can additionally pass a high-quality held-out test suite [86, 114, 145].
- **Correct patch** is a generalizable patch that adheres to the specification of the program. For nontrivial programs in practice, specifications are often hard to obtain and thus tests are used for approximation. By definition, a generalizable patch that can pass all *possible* tests is a correct patch. It is important to note that existing work often defines correct patch differently, as a patch that is syntactically or semantically equivalent to the developer patch via manual check. We argue that checking syntactic and semantic equivalence is hard to automate and can be subjective. For Defects4J, the distinction between generalizable patch and correct patch is not important because we lack the absolute correctness ground truth, which can be approximate with the developer patch but the process of manual checking can be noisy. For this reason, we did not attempt to classify correct patches for Defects4J and only discuss generalizable patches, for which we can automate objectively.

Using this taxonomy, we take multiple measures to evaluate patch quality. For IntroClassJava, the specifications are simple enough that we can verify patch correctness via symbolic execution and constraint solving. We use our customized symbolic execution verifier to identify *correct patches* for IntroClassJava except for `digits`. We treat `digits` differently because the buggy programs often involve loops that make our symbolic execution engine slow. Given the sheer number of plausible patches we can identify for `digits`, we limit each program input to $[-100, 100]$. We set this range based on our observation of different `digits` bugs. Because of this additional constraint, we only identify *generalizable patches* for `digits`.

For Defects4J, we follow existing practice to use an additional held-out test suite to identify *generalizable patches* [86, 114, 145]. We reuse existing high-quality held-out tests that were specifically constructed for evaluating patch quality [114]. For the few bugs that we cannot find existing held-out tests, we manually examine the generated patch and mark it as *generalizable* only if it is *syntactically* or *semantically* close to the developer patch.

Note that by default, VarFix outputs *minimized* plausible patches, as discussed in Section 5.6. Since IntroClassJava bugs are simple enough to reliably verify patch correctness using symbolic execution, we take this opportunity to study how minimization affects patch quality. To that end, we modify VarFix to record all plausible patches and subsequently verify each of them using symbolic execution. But for Defects4J, we record only *minimized plausible patches* due to the sheer quantify of plausible patches available in the large search spaces.

Table 5.4: Comparing the number of different kinds of patches for IntroClassJava.

| | VarFix | | | | | | GenProg | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Subject | $\text{Bug}_{pl}$ | $\text{Bug}_{crt}$ | $\overline{\text{Pl}}$ | $\overline{\text{M-Pl}}$ | $\overline{\text{Crt}}$ | $\overline{\text{M-Crt}}$ | $\text{Bug}_{pl}$ | $\text{Bug}_{crt}$ | $\overline{\text{Pl}}$ | $\overline{\text{M-Pl}}$ | $\overline{\text{Crt}}$ | $\overline{\text{M-Crt}}$ |
| `median` | 32 | 23 | 3261 | 42 | 3222 | 23 | 18 | 13 | 77 | 4 | 79 | 3 |
| `smallest` | 38 | 15 | 2423 | 102 | 219 | 7 | 18 | 2 | 42 | 3 | 3 | 1 |
| `grade` | 4 | 4 | 102 | 4 | 99 | 4 | 4 | 4 | 44 | 2 | 41 | 2 |
| `checksum` | 2 | - | 4 | 4 | - | - | 1 | - | 22 | 2 | - | - |
| `syllables` | 1 | - | 105 | 7 | - | - | 1 | - | 18 | 2 | - | - |
| `digits`* | 30 | 22 | 280 | 9 | 238 | 6 | 26 | 16 | 229 | 4 | 140 | 3 |
| $\texttt{median}_3$ | 18 | 7 | 1944 | 34 | 4377 | 28 | 14 | 7 | 158 | 4 | 274 | 5 |
| $\texttt{smallest}_3$ | 30 | 4 | 3908 | 310 | 1908 | 44 | 20 | 3 | 74 | 3 | 23 | 1 |
| $\texttt{grade}_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\texttt{checksum}_3$ | 2 | - | 4 | 4 | - | - | 1 | - | 64 | 2 | - | - |
| $\texttt{syllables}_3$ | 1 | - | 12 | 12 | - | - | 1 | - | 93 | 4 | - | - |
| $\texttt{digit}_3^*$ | 12 | 8 | 135 | 10 | 164 | 9 | 12 | 7 | 255 | 6 | 144 | 3 |

$\text{Bug}_{pl}$ reports the number of bugs fixed by *plausible* patches.
$\text{Bug}_{crt}$ reports the number of bugs fixed by *correct* patches.
$\overline{\text{Pl}}$ reports the average number of *plausible* patches.
$\overline{\text{M-Pl}}$ reports the average number of *minimized plausible* patches.
$\overline{\text{Crt}}$ reports the average number of *correct* patches.
$\overline{\text{M-Crt}}$ reports the average number of *minimized correct* patches.
A hyphen (-) means we cannot use our customized symbolic execution engine to verify the subject due to technical limitations.
Subjects with the "3" subscript will be discussed in RQ3 (Fixing Ingredients).
`digits` and $\texttt{digits}_3$ are separated from other subjects because we only verify whether a patch is *generalizable*, instead of *correct* (as discussed in the experiment design of RQ2 (Patch Quality)).

**Threats to Validity.** Assessing patch quality remains an important open challenge in automatic program repair. To avoid bias, we use objective and established measures when possible, by relying on independent held-out tests and our customized symbolic execution engine. We had to discard a small fraction of held-out tests for Closure that we cannot reliably execute, due to issues related to the virtual file system of Evosuite. Our prototype symbolic execution engine failed to analyze 8 IntroClassJava bugs due to the use of unsupported language features (e.g., array) or APIs (e.g., `java.util.Scanner#findInLine`). For these bugs, we conservatively mark all patches as just *plausible patches*. For the few Defects4J bugs where manual checking is unavoidable, we identify *generalizable patches* conservatively and present them as examples for open investigation.

**Results (IntroClassJava).** In Table 5.4, we present the average number of generated patches. Details of individual cases are available in Table 5.9 and Table 5.10. We can see that VarFix can find high-quality patches for a large percentage of repaired bugs (columns $\text{Bug}_{pl}$ and $\text{Bug}_{crt}$), ranging from 39.5% for `smallest` to 100% for `grade`. When compared to GenProg, VarFix can fix all the bugs that GenProg can fix while fixing 28 more bugs with high quality patches. These results indicate that VarFix is effective at identifying high-quality patches that are available in the search space.

Moreover, we can see that plausible patches (column $\overline{\text{Pl}}$) and correct patches (column $\overline{\text{Crt}}$) are abundant even for tiny programs like the ones in IntroClassJava. Long and Rinard [96]

analyzed the search spaces of 24 bugs and discovered that plausible patches are common but correct patches are rare. Our results agree with their study that plausible patches are abundant, but also reveal that correct patches can be abundant as well. We speculate that Long and Rinard [96] could not find a large number of correct patches because their search strategies are not effective enough to identify the rest of the correct patches. A direct comparison with the work of Long and Rinard [96] is difficult to set up because their subjects are written in C. However, we can use GenProg to approximate the search strategies of Long and Rinard [96], given that they are all based on search heuristics and we gave GenProg a generous search budget in the same spirit as the experiment design of Long and Rinard [96]. As we can see from Table 5.4, VarFix can find significantly more correct patches than GenProg, indicating that high-quality patches can be abundant, and VarFix is a much more reliable technique for identifying them.

**Results (Effect of Minimization).** When analyzing the effect of minimization (columns $\overline{\text{M-Pl}}$ and $\overline{\text{M-Crt}}$ in Table 5.4), we find that minimization can effectively reduce the number of patches, of both low quality and high quality. For example, minimization can reduce plausible patches to two orders of magnitude fewer for all subjects except for `checksum`. To study the effect of minimization on patch quality, we sampled the patches that were reduced by minimization and manually checked them.

In Figure 5.5, we show an example where minimization is useful for improving patch quality. The buggy program tries to find the median number of three input integers `a`, `b`, and `c`. The root cause of this bug is that the second `if` statement overrules the first one, rendering it useless and so the buggy program can never tell that `a` is the median. The most straightforward fix to this bug is probably changing the `else` in Line 4 to `else if`. However, this change is beyond the scope of our fixing ingredients, so VarFix came up with a workaround, marked as Edit 1 in Figure 5.5. Edit 1 appends the first `if` statement to the end, so that `a` can now be checked without being overruled. Edit 1 alone is sufficient to pass all provided tests, and in fact forms a *correct patch*. Based on this edit, VarexC was able to find a lot more multi-edit patches, such as by swapping out the useless `if` statement in Line 1-3 with other statements that do not affect the eventual program state. Despite being correct, these patches are of little value and might even be considered noise from the code maintenance perspective. This example shows that minimization can be helpful in eliciting high-quality patches.

However, there are also cases where minimization could preemptively remove valuable patches. We show such an example in Figure 5.6. As we can see, the use of relational operators is wrong and so the buggy program cannot handle cases where `a`, `b`, and `c` could be equal. To fix this problem, VarexC generated a three-edit patch as shown in Figure 5.6. Edit 3 replaces the last `if` statement with its body, effectively changing the last `else if` branch into a `else` branch that always considers `c` as the median. With only Edit 3, the program would still fail for cases where `a` and `b` are equal, and thus we need Edit 1 and Edit 2 to fully fix the buggy program. Although Edit 1-3 together form a correct patch, VarFix would not generate such a patch by default due to minimization. As discussed in Section 5.6, VarFix only outputs patches that are minimally sufficient to pass all provided tests, to avoid spamming developers with cases like the one discussed in Figure 5.5. If we look at the provided tests at the bottom of Figure 5.6, we can see that there is no such case where `a == b && b > c`, and thus Edit 1 is not necessary to pass

```
if ((a >= b && a <= c) || (a >= c && a <= b)) {
    median = a;
}
if ((b >= a && b <= c) || (b >= c && b <= a)) {
    median = b;
} else {
    median = c;
1 if ((a >= b && a <= c) || (a >= c && a <= b)) {
1   median = a;
1 }
}
```

Figure 5.5: Code snippet modeled after `median-0cdfa335-003` in the IntroClassJava dataset.

```
if ((a > b && b > c) || (c > b && b > a))
    median = b;
else if ((b 1 > ⇒ >= a && a > c) || (c > a && a 2 > ⇒ >= b))
    median = a;
else 3 if ((a > c && c > b) || (b > c && c > a))
    median = c;

Blackbox
(2,6,8)→6, (2,8,6)→6, (6,2,8)→6, (6,8,2)→6, (8,2,6)→6, (8,6,2)→6, (9,9,9)→9

Whitebox
(0,0,0)→0, (2,0,1)→1, (0,0,1)→0, (0,1,0)→0, (0,2,1)→1, (0,2,3)→2
```

Figure 5.6: Code snippet modeled after `median-3cf6d33a-007` in the IntroClassJava dataset. Blackbox and Whitebox are the names of the test suites that come with the buggy program. Each test case is represented in the form (a, b, c) → o where a, b, c are inputs to the test and c is the expected output.

all provided tests. For this reason, VarFix will only generate a patch with Edit 2 and Edit 3 and miss the correct patch if we had not configured VarFix to record all plausible patches for this experiment.

As we can see, there are cases where minimization is helpful for improving patch quality (e.g., Figure 5.5) and (potentially rare) cases where we might miss certain high-quality patches due to minimization and weak specification (e.g., Figure 5.6). Quantifying which cases are more common than other is unfortunately difficult because manually checking tens of thousands of patches is time-consuming. Based on our observation, cases like Figure 5.6 are relatively rare, so we decided to apply minimization by default.

**Results (Defects4J).** In Table 5.5, we show different kinds of patches VarFix and GenProg generated. As we can see, VarFix can find a nontrivial number of *minimized plausible patches*, even for large programs like Math and Closure with thousands of tests. For 11 out of 24 Math bugs and 6 out of 11 Closure bugs, VarFix can find high-quality *generalizable patches*. In most cases, the number of generalizable patches is small, likely because we have limited fixing ingredients in the search space. When compared to GenProg, VarFix can consistently find more *minimized plausible* and *generalizable* patches, indicating that the search strategy of VarFix is more effective and reliable.

Table 5.5: Different kinds of patches for Math and Closure.

| | Developer Patch | | M-Pl | | Gen | | |
|---|---|---|---|---|---|---|---|
| Bug | Possible? | Generated? | GenProg | VarFix | GenProg | VarFix | Order |
| Math-5 | ✓ | ✓ | 1 | 1 | 1 | 1 | 1/0/0 |
| Math-8 | ✗ | | 3 | 14 | 0 | 0 | |
| Math-22 | ✓ | ✓ | 0 | 2 | 0 | 1 | 0/1/0 |
| Math-24 | ✗ | | 0 | 1 | 0 | 0 | |
| Math-28 | ✗ | | 32 | 69 | 0 | 0 | |
| Math-29 | ✗ | | 1 | 4 | 0 | 0 | |
| Math-35 | ✗ | | 2 | 4 | 1 | 2 | 0/1/1 |
| Math-40 | ✗ | | 1 | 9 | 0 | 0 | |
| Math-49 | ✗ | | 5 | 6 | 0 | 0 | |
| Math-50 | ✓ | ✗ | 11 | 30 | 7 | 16 | 16/0/0 |
| Math-53 | ✓ | ✓ | 2 | 2 | 2 | 2 | 2/0/0 |
| Math-56 | ✗ | | 0 | 3 | - | - | |
| Math-62 | ✗ | | 0 | 3 | 0 | 3 | 0/3/0 |
| Math-65 | ✗ | | 1 | 1 | - | - | |
| Math-70 | ✓ | ✓ | 3 | 3 | 2 | 2 | 2/0/0 |
| Math-73 | ✗ | | 0 | 2 | 0 | 0 | |
| Math-80 | ✗ | | 0 | 6 | 0 | 0 | |
| Math-81 | ✗ | | 0 | 19 | 0 | 1 | 1/0/0 |
| Math-82 | ✓ | ✓ | 1 | 6 | 1 | 6 | 4/2/0 |
| Math-84 | ✗ | | 4 | 5 | 0 | 0 | |
| Math-85 | ✓ | ✓ | 11 | 28 | 2 | 4 | 2/2/0 |
| Math-88 | ✗ | | 0 | 1 | 0 | 1 | 0/0/1 |
| Math-95 | ✗ | | 10 | 13 | 0 | 0 | |
| Math-96 | ✗ | | 1 | 1 | - | - | |
| Closure-11 | ✓ | ✗ | 0 | 1 | 0 | 1 | 1/0/0 |
| Closure-13 | ✓ | ✗ | 0 | 28 | 0 | 27 | 27/0/0 |
| Closure-19 | ✗ | | 0 | 5 | 0 | 0 | |
| Closure-21 | ✗ | | 0 | 78 | 0 | 0 | |
| Closure-22 | ✗ | | 0 | 97 | 0 | 0 | |
| Closure-62* | ✓ | ✓ | 0 | 2 | 0 | 2 | 2/0/0 |
| Closure-63* | ✓ | ✓ | 0 | 2 | 0 | 2 | 2/0/0 |
| Closure-66 | ✗ | | 0 | 8 | 0 | 0 | |
| Closure-73 | ✓ | ✓ | 0 | 1 | 0 | 1 | 1/0/0 |
| Closure-86 | ✓ | ✓ | 0 | 1 | 0 | 1 | 1/0/0 |
| Closure-126 | ✓ | ✗ | 4 | 9 | 0 | 0 | |
| Closure-161 | ✓ | ✗ | 0 | 1 | 0 | 1 | 1/0/0 |

A hyphen (-) denotes missing data.
"Possible?" denotes whether it is possible to use our mutation operators to generate the developer patch.
"Generated?" denotes whether VarFix generated the developer patch in our evaluation.
M-Pl denotes the number of *minimized plausible patches*.
Gen denotes the number of *generalizable patches*.
Order breaks down VarFix's generalizable patches (second to last column) by order (i.e., number of edits), from order 1 to order 3.
Closure-62 and Closure-63 are duplicate. We show them in this table for consistency with prior work, but only count one of them when we report numbers [139].

```
1  if (childType.isDict()) {
2      report(t, property, TypeValidator.ILLEGAL_PROPERTY_ACCESS, "'.'", "dict");
3  1 } else if (n.getJSType() != null && parent.isAssign()) {
4  1     return;
5  } else if (validator.expectNotNullOrUndefined(t, n, childType,
6      "No properties on this expression", getNativeType(OBJECT_TYPE))) {
7      checkPropertyAccess(childType, property.getString(), t, n);
```

(a) Developer patch

```
1  if (childType.isDict()) {
2      report(t, property, TypeValidator.ILLEGAL_PROPERTY_ACCESS, "'.'", "dict");
3  } else if (n.getJSType() 1 != ⇒ == null && parent.isAssign()) {
4      return;
5  } else if (validator.expectNotNullOrUndefined(t, n, childType,
6      "No properties on this expression", getNativeType(OBJECT_TYPE))) {
7      checkPropertyAccess(childType, property.getString(), t, n);
```

(b) Generalizable patch generated by VarFix

Figure 5.7: Developer patch and VarFix patch for Closure-11.

When compared to developer patches, VarFix can generate the developer patch for 6 Math bugs and 4 Closure bugs. For the few cases where the developer patch is feasible but not generated, we manually examined the corresponding meta-programs to look for the fixing ingredients needed to generate the developer patch. It turns out that in all cases (i.e., Math-50, Closure-11, Closure-13, Closure126, and Closure-161), at least some of the required fixing ingredients are missing in the meta-programs. We argue that this is the limitation of GenProg, which we use to perform fault localization and generate individual edits, and thus independent of the main contribution of this work. Although VarFix cannot generate the developer patch in these cases, VarFix can still generate high-quality *generalizable patches*. We discuss a few concrete examples below.

In Figure 5.7, we compare the developer patch and the patch generated by VarFix for `Closure-11`. Although not syntactically the same, they are semantically close because both patches try to skip the `else if` branch in Line 3-4. Similarly, in Figure 5.8, we contrast the developer patch and the VarFix patch for `Math-50`. Instead of removing the `if` statement like the developer patch does, the VarFix patch replaces it with a return check that is taken from the very beginning of the surrounding method. Since the return check appears much earlier in the method, Line 8-10 is unlikely to take effect. Looking at the code statically, there exist execution paths that modify `f0` between the original return check and the copied return check introduced by the VarFix patch, but more domain knowledge is needed to determine whether those paths are feasible. Again, albeit being syntactically different, the two patches in Figure 5.8 is likely to be semantically close. The fact that the VarFix patches in Figure 5.7 and Figure 5.8 can pass an independent held-out suite indicates that the patches have high quality.

Finally, in Figure 5.9, we show an example where the VarFix patch is nearly semantically identical to the developer patch. Both patches add the same return check, but in two different methods of the same class. A closer look at the code reveals that the method `tryFoldArrayAccess`, which is modified in the developer patch, is only called in method `tryFoldGetElem`, which is modified in the VarFix patch. Given the fact that both methods are `private`, the two patches

```
    break;
case REGULA_FALSI:
    // Nothing.
-   if (x == x1) {
-     x0 = 0.5 * (x0 + x1 - FastMath.max(rtol * FastMath.abs(x1), atol));
-     f0 = computeObjectiveValue(x0);
-   }
    break;
default:
    // Should never happen.
```

(a) Developer patch

```
    break;
case REGULA_FALSI:
    // Nothing.
-   if (x == x1) {
-     x0 = 0.5 * (x0 + x1 - FastMath.max(rtol * FastMath.abs(x1), atol));
-     f0 = computeObjectiveValue(x0);
-   }
+   if (f0 == 0.0) {
+     return x0;
+   }
    break;
default:
    // Should never happen.
```

(b) Generalizable patch generated by VarFix

Figure 5.8: Developer patch and VarFix patch for Math-50.

are semantically very close. We refrain from asserting semantically equivalence because it is theoretically possible that `tryFoldGetElem` is called outside of the Java class using reflection.

From these examples, we can see that VarFix is effective at generating high-quality patches that resemble the developer patch. They also illustrate that there could be different ways of fixing a bug, and thus using the developer patch as the only gold standard could be limited, and even infeasible since determining semantic equivalence is often difficult for nontrivial programs.

**Summary**: RQ2 (Patch Quality)

To what extent do our generated patches overfit to the provided tests?

- For both IntroClassJava and Defects4J, VarFix can identify significantly more high-quality patches for more bugs than GenProg, suggesting that the systematic search is useful for identifying high-quality patches available in the search space.
- Minimization is useful for filtering out trivial patches, but can also occasionally exclude high-quality patches that are not distinguishable by the test suite.

```
1  private Node tryFoldArrayAccess(Node n, Node left, Node right) {
2      Node parent = n.getParent();
3  1   if (isAssignmentTarget(n)) {
4  1     return n;
5  1   }
```

(a) Developer patch

```
1  private Node tryFoldGetElem(Node n, Node left, Node right) {
2      Preconditions.checkArgument(n.getType() == Token.GETELEM);
3  1   if (isAssignmentTarget(n)) {
4  1     return n;
5  1   }
6      if (left.getType() == Token.OBJECTLIT) {
7          return tryFoldObjectPropAccess(n, left, right);
8      }
9      if (left.getType() == Token.ARRAYLIT) {
10         return tryFoldArrayAccess(n, left, right);
11     }
12     return n;
13 }
```

(b) Generalizable patch generated by VarFix

Figure 5.9: Developer patch and VarFix patch for Closure-161.

### 5.8.6 RQ3 (Fixing Ingredients)

**Experiment Setup.** As discussed in RQ1 (Effectiveness) and RQ2 (Patch Quality), our approach is effective at generating high-quality patches. We hypothesize that VarFix is effective because the systematic search enabled by variational execution can effectively make use of the diverse fixing ingredients in the search space. In this research question, we validate our hypothesize by simulating different fixing ingredients in the search space and explore whether the effectiveness of our approach is affected. To that end, we design three experiments.

First, for IntroClassJava, we use the original three edit templates of GenProg to regenerate meta-programs. These new meta-programs only have edits that append, replace, or delete statements. In contrast, the meta-programs generated with all eight edit templates (Section 5.4) have more diverse fixing ingredients—expression-level edits that modify branch conditions. We repeat the experiment setup of RQ1 (Effectiveness) to run VarFix and GenProg on these new meta-programs and compare effectiveness of both approaches. We suspect that VarFix can make use of the more diverse fixing ingredients in the search space to generate more high-quality patches for more bugs. Moreover, we expect a similar improvement with GenProg, but suspect that VarFix's improvement is bigger.

Second, for Defects4J, we manually examine all bugs to gauge whether the developer patch can be simulated using our eight edit templates, and more importantly, why VarFix failed to fix the bug. This manual analysis helps us to understand if there is a case where VarFix failed to make use of the available fixing ingredients in the search space.

Finally, we pick 10 bugs that have not been fixed by any prior work and manually enhance the meta-programs with fixing ingredients of their developer patch. To make the search more challenging, we focus on bugs that need two or three edits in their developer patch. This way,

the correct patch is guaranteed to be in the search space and an effective approach should be able to identify it. We run VarFix and GenProg with the same settings as RQ1 (Effectiveness) and compare the number of bugs fixed this way.

**Threats to Validity.** In addition to the threats discussed in RQ1 (Effectiveness), our manual selection of 10 Defects4J bugs might not be generalizable. We argue that manually enhancing meta-programs is time-consuming, so we limit the scale of this experiment, but make it more challenging by picking bugs that require multiple edits to fix and have not been fixed by any existing work. There is also a question of whether the fixing ingredients of developer patches can be generated with automated techniques. We argue that this work focuses on the new search strategy and thus generating fixing ingredients is beyond the scope of this work.

**Results (IntroClassJava).** In the lower half of Table 5.4, we report the number of bugs fixed and the number of patches generated. Again, details of individual cases are available in Table 5.9 and Table 5.10.

We can see that VarFix can fix more bugs for all subjects, both plausibly (column $\text{Bug}_{pl}$) and correctly (column $\text{Bug}_{crt}$). For example for `median`, VarFix can generate correct patches for 16 more bugs when all eight edit templates are used to generate more diverse fixing ingredients. GenProg can also make use of the fixing ingredients, but to a lesser extent than VarexC. For example, even though using the same meta-programs of `median`, GenProg can only generate correct patches for 6 more `median` bugs. Moreover, GenProg fixed 2 fewer `smallest` bugs when eight edit templates are used, indicating that the randomness of genetic search can potentially undermine the improvement of diverse fixing ingredients.

If we compare the number of generated patches, we can observe two opposite trends. When switched to using 8 edit templates, the number of patches can both increase (e.g., from 1944 to 3261 for `median`) and decrease (e.g., from 3908 to 2423 for `smallest`). But there is a clear trend that fewer *minimized correct patches* (column $\overline{\text{M-Crt}}$) are available in the search spaces constructed with eight edit templates. GenProg exhibits similar trends as VarFix, but at a smaller scale. We speculate that, using more edit templates can increase the likelihood of a bug being fixed, but can also decrease the number of patches generated if certain important fixing ingredients are squeezed out. Recall that, we set a limit of 500 edits independent of the edit templates used, and thus we can only keep 500 edits that GenProg considers most likely useful. We could increase the bound to include more edits in the search space, but we argue that variational execution is not a silver bullet to the combinatorial explosion problem. Instead of pushing variational execution or other similar techniques to handle arbitrary search spaces of any sizes, we argue that it is more viable to improve techniques of generating fixing ingredients to form a search space that contains valuable fixing ingredients.

**Results (Defects4J).** We manually examined all developer patches of Math and Closure and discovered that 9 more bugs could have been fixed using our eight edit templates. A closer look into the corresponding meta-programs revealed that none of them contains *all* required fixing ingredients of the developer patch. There are two causes to missing required fixing ingredients. First, the fault localization used in GenProg failed to identify the buggy locations, the problem

Table 5.6: Patch generalization for bugs with manually encoded fixing ingredients.

| Bug | Edits | Methods | Classes | VarFix | GenProg |
|---|---|---|---|---|---|
| Math-1 | 2 | 2 | 2 | ✓ | ✓ |
| Math-26 | 2 | 1 | 1 | ✓ | ✓ |
| Math-52 | 3 | 1 | 1 | ✓ | ✓ |
| Math-83 | 3 | 2 | 1 | ✓ | ✗ |
| Math-86 | 2 | 1 | 1 | ✓ | ✓ |
| Closure-7 | 2 | 1 | 1 | ✓ | ✗ |
| Closure-9 | 2 | 2 | 2 | ✓ | ✓ |
| Closure-27 | 3 | 2 | 1 | ✓ | ✗ |
| Closure-72 | 2 | 2 | 2 | ✓ | ✗ |
| Closure-75 | 2 | 2 | 1 | ✓ | ✗ |

Edits denotes the number edits in the developer patch.
Methods denotes the number of methods modified in the developer patch.
Classes denotes the number of classes modified in the developer patch.

of which exists in Math-46, Math-72, Closure-133, and Closure-150. Second, GenProg does not consider the required fixing ingredients among the most 5000 useful edits (recall that we limit the number of generated edits to 5000 for Defects4J), the problem of which exists in Math-69, Closure-102, Closure-115, Closure-117, and Closure-126. These results together with Table 5.5 indicate that VarFix has already generated all developer patches that are available in the search space, and thus our approach as a search strategy is effective. Fault localization and meta-program generation can be improved to fix more bugs, but this work focuses on the search strategy.

In Table 5.6, we show statistics of the bugs and their corresponding developer patch. 7 of them require 2 edits to fix and 3 of them require 3 edits. Most of the required edits span across 2 methods, and in three bugs even span across 2 classes. As we can see, VarFix can identify the developer patch for all manually enhanced search spaces, while GenProg missed the developer match in half of the cases. Interestingly, in 4 out of 10 cases, VarFix found a patch that contains fewer edits than the developer patch, indicating that not all edits in the developer patch are necessary for passing all provided tests. In 2 out of 10 cases, VarFix found more minimized plausible patches, indicating that some edits of the developer patch can be swapped out with the automatically generated fixing ingredients. These findings further suggest that using developer patches as the only standard for evaluating patch quality could be limited.

**Summary**: RQ3 (Fixing Ingredients)

To what extent can our approach make use of different kinds of fixing ingredients?

Using a systematic search, VarFix can effectively make use the necessary fixing ingredients in the search space to fix more bugs whereas heuristics-based approach like GenProg can sometimes miss fixing ingredients due to the ad hoc exploration of search space.

Table 5.7: VarFix patches for IntroClassJava by order.

| | | Patched By | | | Minimized Correct Patches | | | |
|---|---|---|---|---|---|---|---|---|
| Subject | $Bug_{crt}$ | O1 | O2 | O3 | Total | O1 | O2 | O3 |
| median | 23 | 7 | 14 | 18 | 537 | 18 | 87 | 432 |
| smallest | 15 | 1 | 3 | 14 | 116 | 1 | 9 | 106 |
| grade | 4 | 4 | 2 | 0 | 18 | 10 | 8 | 0 |
| checksum | - | - | - | - | - | - | - | - |
| syllables | - | - | - | - | - | - | - | - |
| digits* | 21 | 14 | 20 | 2 | 135 | 29 | 89 | 17 |

A hyphen (-) denotes missing data.
$Bug_{crt}$ denotes the number of bugs that can be fixed by *correct* patches.
O1, O2, O3 represents first-order, second-order, third-order patch, respectively.
For digits, we discuss *generalizable* patches as opposed to *correct* patches because of the limitations in our customized symbolic execution engine.

### 5.8.7 RQ4 (Multi-Edit)

**Experiment Setup.** To understand how effective our approach is at generating multi-edit patches, we count the bugs for which VarFix can generate correct patches composed of 1 edit, 2 edits, and 3 edits. We also break down all minimized correct patches by order to observe the distribution. For Defects4J, we include the 10 bugs with manually enhanced meta-programs from RQ3 (Fixing Ingredients) because they feature multi-edit patches. We focus on *minimized correct patches* because they have the highest quality among the patches generated by VarFix, excluding unnecessary multi-edit patches that can be easily filtered out by patch minimization.

**Threats to Validity.** This research question shares the same threats as previously discussed.

**Results (IntroClassJava).** In Table 5.7, we show the collective results for IntroClassJava. Details of individual bugs can be found in Table 5.9 and Table 5.10. As we can see, the number of bugs fixed correctly would decrease significantly if only one edit is used. Between second-order and third-order, there is no clear trend of which is more common, likely because the number of edits needed depends on multiple factors, such as the nature of the bug and the fixing ingredients available in the search space. If we look at the breakdown of *minimized correct patches*, we can see that the number of first-order patches is relatively small comparing to multi-edit patches. Together with the results reported in Table 5.2 of RQ1 (Effectiveness), these results suggest that VarFix's search strategy is effective at composing multiple edits to generate more high-quality patches to fix more bugs.

**Results (Defects4J).** In Table 5.5, we show a breakdown of identified *generalizable patches* by order. Among the 17 bugs for which VarFix can generate generalizable patches, 4 of them require at least two edits. Overall, the number of multi-edit patches is smaller than single-edit patches. We speculate the main reason to be lack of proper fixing ingredients in the search space, the problem of which is orthogonal to the main contributions of this work. With all necessary fixing ingredients, VarFix can effectively gather all necessary edits to form a patch, such as the

```
// org/apache/commons/math3/distribution/FDistribution.java
public boolean isSupportLowerBoundInclusive() {
1   return true;
1   return false;
}

// org/apache/commons/math3/distribution/UniformRealDistribution.java
public boolean isSupportUpperBoundInclusive() {
1   return false;
1   return true;
}
```

Figure 5.10: A multi-edit patch generated by VarFix for `Math-22`. This patch is the same as the developer patch, and we are the first to report fixing this particular bug.

one shown in Figure 5.10, where two `return` statements in two classes need to be modified. To the best of our knowledge, we are the first to generate a high quality patch for this bug. The results of the 10 manually enhanced meta-programs in Table 5.6 further suggest that VarFix's search strategy is effective at identifying multi-edit patches that are available in the search space, even at a larger scale than IntroClassJava.

**Summary**: RQ4 (Multi-Edit)

How effective is our approach in generating multi-edit patches?

For both IntroClassJava and Defects4J, VarFix can systematically explore the search space to gather necessary fixing ingredients to form multi-edit patches, thus fixing more bugs and leading to more high-quality patches than GenProg.

### 5.8.8 RQ5 (Patch Ranking)

**Experiment Setup.** To evaluate our patch ranking, we compare our ranking strategies with two baselines on a subset of IntroClassJava bugs. As discussed in Section 5.6, VarFix by default outputs *minimized plausible patches*. We apply our patch ranking mechanism to these *minimized plausible patches*, for a subset of IntroClassJava bugs that (1) have at least one *correct patch* and (2) have at least one plausible but incorrect patch. The first criterion ensures that we have a ground truth to evaluate our ranking. The second criterion excludes cases where ranking is not necessary because all patches are correct. We do not rank patches of Defects4J for two reasons. First, patch correctness as the ground truth of ranking is difficult to establish for large programs like Math and Closure, making it difficult to objectively evaluate the ranking results. Second, unlike IntroClassJava where we can identify many minimized patches, the number of minimized patches as shown in Table 5.5 are small for Math and Closure and thus a ranking is not interesting or necessary.

As baselines, we measure AST-based syntactic distance and Levenshtein distance. Researchers have proposed different strategies for ranking *patch candidates*. Xiong et al. [168] systematically analyzed existing strategies and discovered that most of them target specific

patch categories (e.g., expression level patches) and thus cannot be applied to general patches. We use the AST-based syntactic distance originally proposed by Le et al. [86], which is defined as the number of AST node changes needed to transform the original program to the patched version. AST node changes are computed using GumTree [38]. The assumption behind such a syntactic distance is to prioritize patches that are close to the original program. Following the same intuition, we also use the Levenshtein distance as a baseline by treating the original and patched programs as strings.

**Threats To Validity.** This experiment shares the same threats as previously discussed.

**Results.** As we can see in Table 5.8, our ranking techniques significantly outperform the baselines, suggesting that the dynamic information obtained from variational execution is useful for distinguishing high-quality patches. There are a few cases where the two baselines (e.g., `smallest-3b2376ab-008`) perform reasonably well, indicating that static information can potentially be combined with dynamic information to further improve patch ranking. CF-line performs the best overall, ranking at least one correct patch to top 10 for all bugs except for one case where the first correct patch is ranked to the 11th place. We suspect that the success of CF-line is due to the use of more fine-grain information. Future research in this direction should explore more fine-grained runtime information for distinguishing high-quality patches.

**Summary**: RQ5 (Patch Ranking)

How effective is our patch ranking?

Our patch ranking strategies based on dynamic information are effective at ranking high-quality patches to the top, significantly outperforming the two baselines that use static information.

Table 5.8: Patch Ranking Results

| Bug | Minimized Plausible Patches | Minimized Correct Patches | AST | Leven. | DF | CF-if | CF-line |
|---|---|---|---|---|---|---|---|
| median-1bf73a9c-003 | 111 | 3 | 48 | 54 | 3 | 8 | 2 |
| median-317aa705-002 | 154 | 150 | 1 | 1 | 1 | 1 | 1 |
| median-36d8008b-000 | 103 | 3 | 39 | 36 | 11 | 6 | 3 |
| median-6aaeaf2f-000 | 103 | 3 | 63 | 39 | 4 | 6 | 2 |
| median-6e464f2b-003 | 52 | 4 | 27 | 21 | 3 | 6 | 2 |
| median-b6fd408d-000 | 5 | 4 | 1 | 1 | 1 | 1 | 1 |
| median-b6fd408d-001 | 16 | 15 | 1 | 1 | 1 | 1 | 1 |
| median-cd2d9b5b-010 | 118 | 6 | 40 | 41 | 2 | 8 | 3 |
| median-d43d3207-000 | 52 | 4 | 26 | 26 | 10 | 9 | 2 |
| median-fcf701e8-002 | 5 | 4 | 2 | 2 | 1 | 2 | 2 |
| median-fcf701e8-003 | 6 | 1 | 4 | 4 | 3 | 4 | 4 |
| smallest-15cb07a7-007 | 84 | 1 | 8 | 7 | 14 | 28 | 5 |
| smallest-1b31fa5c-003 | 146 | 3 | 19 | 7 | 21 | 47 | 11 |
| smallest-346b1d3c-010 | 59 | 24 | 1 | 1 | 1 | 1 | 1 |
| smallest-36d8008b-003 | 207 | 2 | 13 | 12 | 11 | 48 | 6 |
| smallest-3b2376ab-008 | 92 | 2 | 4 | 4 | 17 | 35 | 9 |
| smallest-48b82975-001 | 207 | 2 | 13 | 12 | 7 | 47 | 6 |
| smallest-68eb0bb0-000 | 63 | 2 | 11 | 11 | 19 | 25 | 8 |
| smallest-90834803-005 | 40 | 2 | 12 | 16 | 5 | 22 | 8 |
| smallest-97f6b152-003 | 145 | 2 | 4 | 4 | 5 | 20 | 1 |
| smallest-cb243beb-000 | 63 | 3 | 6 | 16 | 1 | 1 | 1 |
| smallest-dedc2a7c-000 | 57 | 2 | 3 | 3 | 15 | 21 | 10 |
| smallest-ea67b841-003 | 142 | 2 | 10 | 10 | 12 | 9 | 10 |
| smallest-f8d57dea-000 | 92 | 2 | 4 | 4 | 4 | 28 | 3 |
| Top 1 | | | 4 | 4 | 6 | 5 | 6 |
| Top 5 | | | 10 | 10 | 14 | 7 | 16 |
| Top 10 | | | 13 | 13 | 16 | 14 | 23 |

Numbers in the last 5 columns denote the rank of the first correct patch in the ranking.
Top $n$ shows the number of bugs for which the correct patch is ranked to top $n$.
AST denotes the baseline that measures GumTree edit distance (see experiment design of RQ5 (Patch Ranking)).
Leven. denotes the baseline that measures Levenshtein distance (see experiment design of RQ5 (Patch Ranking)).
DF denotes our patch ranking based on data-flow information (see $distance_{var}$ in Section 5.6).
CF-if denotes our patch ranking based on executed `if` branches (see $distance_{if}$ in Section 5.6).
CF-line denotes our patch ranking based on executed lines (see $distance_{line}$ in Section 5.6).

Table 5.9: Different kinds of patches for IntroClassJava.
3 Mut and 8 Mut represent using 3 mutation operators and 8 mutation operators, respectively.
Pl, Crt, M-Pl, and M-Crt represent the number of plausible, correct, minimized plausible, and minimized correct patches, respectively.
Order breaks down VarFix's minimized correct patches by order, from order 1 to order 3.
Each cell shows the number for **GenProg/VarFix** and a hyphen (-) denotes data unavailable.

| | 3 Mut | | | | 8 Mut | | | |
|---|---|---|---|---|---|---|---|---|
| Bug | Pl | Crt | M-Pl | M-Crt | Pl | Crt | M-Pl | M-Crt (Order) |
| m-0cd-03 | 702/9888 | 702/9888 | 11/69 | 11/69 | 168/7793 | 168/7793 | 13/55 | 13/55 (1/16/38) |
| m-0ce-03 | -/4 | -/0 | -/4 | -/0 | 2/78 | 0/2 | 1/11 | 0/2 (0/0/2) |
| m-1b-00 | 180/1604 | 0/0 | 7/60 | 0/0 | 159/8507 | 2/24 | 16/337 | 1/24 (0/0/24) |
| m-1b-03 | 19/592 | 0/0 | 8/71 | 0/0 | 60/6476 | 11/305 | 4/111 | 3/41 (0/4/37) |
| m-31-00 | 697/713 | 697/713 | 10/23 | 10/23 | 461/33017 | 435/32841 | 8/53 | 8/53 (7/9/37) |
| m-31-02 | 139/280 | 139/280 | 9/19 | 9/19 | 136/11336 | 136/11227 | 5/154 | 5/150 (2/24/124) |
| m-31-03 | -/- | -/- | -/- | -/- | -/8 | -/0 | -/8 | -/0 (0/0/0) |
| m-36-00 | 1/592 | 0/0 | 1/71 | 0/0 | 30/3282 | 2/149 | 5/103 | 1/22 (0/2/20) |
| m-3b-03 | 7/398 | 7/398 | 1/18 | 1/18 | 16/1630 | 16/1630 | 3/4 | 3/4 (2/2/0) |
| m-3b-06 | 49/61 | 49/61 | 1/1 | 1/1 | 5/914 | 5/914 | 1/1 | 1/1 (1/0/0) |
| m-3c-07 | -/4 | -/0 | -/4 | -/0 | -/145 | -/2 | -/16 | -/2 (0/0/2) |
| m-6a-00 | 1/592 | 0/0 | 1/71 | 0/0 | 11/3362 | 0/81 | 3/103 | 0/16 (0/1/15) |
| m-6e-03 | 2/180 | 0/0 | 2/22 | 0/0 | 10/2394 | 0/190 | 2/52 | 0/32 (0/4/28) |
| m-89-10 | 325/19094 | 325/19088 | 7/67 | 7/67 | 207/16775 | 207/16751 | 4/32 | 4/32 (4/0/28) |
| m-90-10 | 4/216 | 4/216 | 1/5 | 1/5 | 7/1175 | 7/1162 | 1/9 | 1/9 (1/8/0) |
| m-90-00 | -/11 | -/0 | -/11 | -/0 | 1/197 | 0/0 | 1/25 | 0/0 (0/0/0) |
| m-93-10 | -/- | -/- | -/- | -/- | -/5 | -/0 | -/5 | -/0 (0/0/0) |
| m-93-12 | -/- | -/- | -/- | -/- | -/214 | -/210 | -/4 | -/4 (0/4/0) |
| m-93-15 | -/- | -/- | -/- | -/- | -/214 | -/210 | -/4 | -/4 (0/4/0) |
| m-aa-03 | -/- | -/- | -/- | -/- | -/4 | -/0 | -/4 | -/0 (0/0/0) |
| m-af-04 | -/- | -/- | -/- | -/- | -/256 | -/0 | -/13 | -/0 (0/0/0) |
| m-af-07 | -/- | -/- | -/- | -/- | -/253 | -/0 | -/13 | -/0 (0/0/0) |
| m-b6-00 | -/- | -/- | -/- | -/- | 0/5 | 0/4 | 0/5 | 0/4 (0/0/4) |
| m-b6-01 | -/- | -/- | -/- | -/- | 39/271 | 39/270 | 1/16 | 1/15 (0/3/12) |
| m-c7-02 | -/- | -/- | -/- | -/- | -/4 | -/0 | -/4 | -/0 (0/0/0) |
| m-cd-10 | 98/592 | 0/0 | 5/71 | 0/0 | 66/3458 | 5/156 | 8/118 | 1/25 (0/2/23) |
| m-d0-00 | -/- | -/- | -/- | -/- | -/8 | -/0 | -/8 | -/0 (0/0/0) |
| m-d43-00 | 1/180 | 0/0 | 1/22 | 0/0 | 20/2394 | 3/190 | 3/52 | 1/32 (0/4/28) |
| m-d4a-00 | -/- | -/- | -/- | -/- | -/4 | -/0 | -/4 | -/0 (0/0/0) |
| m-e9-01 | -/4 | -/0 | -/4 | -/0 | 1/70 | 0/1 | 1/10 | 0/1 (0/0/1) |
| m-fc-02 | -/- | -/- | -/- | -/- | -/38 | -/6 | -/5 | -/6 (0/0/6) |
| m-fc-03 | -/- | -/- | -/- | -/- | -/94 | -/3 | -/6 | -/3 (0/0/3) |
| s-15-07 | -/237 | -/0 | -/38 | -/0 | 103/3025 | 0/4 | 2/84 | 0/4 (0/0/4) |
| s-1b-03 | 13/734 | 0/0 | 3/74 | 0/0 | 55/10177 | 0/3 | 4/146 | 0/3 (0/0/3) |
| s-26-00 | -/- | -/- | -/- | -/- | -/321 | -/0 | -/36 | -/0 (0/0/0) |
| s-30-07 | 1/110 | 0/0 | 1/4 | 0/0 | 133/10022 | 0/6 | 2/26 | 0/6 (0/0/6) |
| s-34-05 | 19/16 | 8/16 | 4/16 | 3/16 | 1/694 | 1/502 | 1/21 | 1/21 (0/6/15) |
| s-34-10 | 111/19 | 33/19 | 3/19 | 1/19 | 5/214 | 0/138 | 1/59 | 0/24 (0/1/23) |
| s-36-03 | 3/1391 | 0/0 | 1/282 | 0/0 | -/6536 | -/8 | -/207 | -/8 (0/0/8) |
| s-3b-07 | 290/34764 | 0/0 | 4/1970 | 0/0 | 28/8549 | 0/0 | 11/498 | 0/0 (0/0/0) |
| s-3b-08 | 43/263 | 0/0 | 2/43 | 0/0 | -/2792 | -/8 | -/92 | -/8 (0/0/8) |
| s-3c-03 | -/- | -/- | -/- | -/- | -/321 | -/0 | -/36 | -/0 (0/0/0) |
| s-48-01 | -/1391 | -/0 | -/282 | -/0 | -/6536 | -/8 | -/207 | -/8 (0/0/8) |

To be continued on next page

**Table 5.9 – continued from preview page**

| | 3 Mut | | | | 8 Mut | | | |
|---|---|---|---|---|---|---|---|---|
| Bug | Pl | Crt | M-Pl | M-Crt | Pl | Crt | M-Pl | M-Crt (Order) |
| s-68-00 | 3/4 | 0/0 | 1/4 | 0/0 | -/2122 | -/4 | -/63 | -/4 (0/0/4) |
| s-6a-01 | 24/13447 | 0/0 | 1/493 | 0/0 | 8/823 | 0/0 | 2/77 | 0/0 (0/0/0) |
| s-76-02 | -/- | -/- | -/- | -/- | -/13 | -/0 | -/13 | -/0 (0/0/0) |
| s-76-03 | -/- | -/- | -/- | -/- | 0/13 | 0/0 | 0/13 | 0/0 (0/0/0) |
| s-76-07 | 1/7789 | 0/0 | 1/1678 | 0/0 | 4/812 | 0/0 | 3/416 | 0/0 (0/0/0) |
| s-76-09 | 11/7789 | 0/0 | 4/1678 | 0/0 | 17/812 | 0/0 | 4/416 | 0/0 (0/0/0) |
| s-76-10 | -/- | -/- | -/- | -/- | 1/13 | 0/0 | 1/13 | 0/0 (0/0/0) |
| s-81-02 | -/6 | -/0 | -/6 | -/0 | -/- | -/- | -/- | -/- (-) |
| s-81-03 | 229/2105 | 0/0 | 2/85 | 0/0 | 2/119 | 0/0 | 1/9 | 0/0 (0/0/0) |
| s-84-07 | -/51 | -/0 | -/51 | -/0 | -/24 | -/0 | -/24 | -/0 (0/0/0) |
| s-88-02 | 115/14475 | 0/0 | 7/498 | 0/0 | 45/5015 | 0/0 | 6/216 | 0/0 (0/0/0) |
| s-88-03 | 73/14475 | 0/0 | 12/498 | 0/0 | 50/5015 | 0/0 | 10/216 | 0/0 (0/0/0) |
| s-88-06 | 80/7 | 0/0 | 1/7 | 0/0 | 3/33 | 0/0 | 1/33 | 0/0 (0/0/0) |
| s-90-00 | -/- | -/- | -/- | -/- | -/1 | -/0 | -/1 | -/0 (0/0/0) |
| s-90-05 | 14/110 | 0/0 | 1/4 | 0/0 | 45/10115 | 0/2 | 2/40 | 0/2 (0/0/2) |
| s-90-01 | -/- | -/- | -/- | -/- | -/1 | -/0 | -/1 | -/0 (0/0/0) |
| s-93-00 | -/31 | -/0 | -/31 | -/0 | -/163 | -/0 | -/45 | -/0 (0/0/0) |
| s-97-03 | -/764 | -/0 | -/135 | -/0 | -/5651 | -/8 | -/145 | -/8 (0/0/8) |
| s-af-00 | 27/828 | 0/140 | 4/828 | 0/140 | -/259 | -/- | -/259 | -/- (-) |
| s-c8-00 | 239/5315 | -/- | 2/23 | -/- | 153/3215 | -/- | 1/7 | -/- (-) |
| s-c9-03 | -/- | -/- | -/- | -/- | -/1 | -/0 | -/1 | -/0 (0/0/0) |
| s-cb-00 | 185/9554 | 28/7458 | 7/201 | 1/1 | 107/3676 | 5/2580 | 6/63 | 2/3 (1/2/0) |
| s-d0-01 | -/51 | -/0 | -/51 | -/0 | -/24 | -/0 | -/24 | -/0 (0/0/0) |
| s-de-00 | -/263 | -/0 | -/43 | -/0 | 1/264 | 0/6 | 1/57 | 0/6 (0/0/6) |
| s-e9-00 | -/- | -/- | -/- | -/- | -/321 | -/0 | -/36 | -/0 (0/0/0) |
| s-ea-03 | 1/985 | 0/0 | 1/209 | 0/0 | -/1428 | -/3 | -/142 | -/3 (0/0/3) |
| s-f2-00 | -/31 | -/0 | -/31 | -/0 | -/163 | -/0 | -/45 | -/0 (0/0/0) |
| s-f8-00 | -/263 | -/0 | -/43 | -/0 | -/2792 | -/8 | -/92 | -/8 (0/0/8) |
| g-b1-01 | -/- | -/- | -/- | -/- | 15/168 | 15/164 | 1/4 | 1/4 (4/0/0) |
| g-b1-03 | -/- | -/- | -/- | -/- | 137/168 | 129/164 | 1/4 | 1/4 (4/0/0) |
| g-bf-00 | -/- | -/- | -/- | -/- | 15/37 | 15/34 | 5/5 | 5/5 (1/4/0) |
| g-bf-01 | -/- | -/- | -/- | -/- | 9/37 | 8/34 | 4/5 | 4/5 (1/4/0) |
| c-36-03 | -/4 | -/- | -/4 | -/- | -/5 | -/- | -/5 | -/- (-) |
| c-e2-05 | 64/4 | -/- | 2/4 | -/- | 22/4 | -/- | 2/4 | -/- (-) |
| sy-ca-03 | 93/12 | -/- | 4/12 | -/- | 18/105 | -/- | 2/7 | -/- (-) |

Table 5.10: Different kinds of patches for `digits`.
3 Mut and 8 Mut represent using 3 mutation operators and 8 mutation operators, respectively.
Pl, Gen, M-Pl, and M-Gen represent the number of plausible, generalizable, minimized plausible, and minimized generalizable patches.
Order breaks down VarFix's minimized generalizable patches by order, from order 1 to order 3.
Each cell shows the number for **GenProg/VarFix** and a hyphen (-) denotes data unavailable.

| | 3 Mut | | | | 8 Mut | | | |
|---|---|---|---|---|---|---|---|---|
| Bug | Pl | Gen | M-Pl | M-Gen | Pl | Gen | M-Pl | M-Gen (Order) |
| d-07-02 | -/- | -/- | -/- | -/- | 423/168 | 423/168 | 4/4 | 4/4 (2/2/0) |
| d-0c-04 | -/- | -/- | -/- | -/- | 159/257 | -/- | 6/10 | -/- (-) |
| d-0c-05 | -/- | -/- | -/- | -/- | 561/120 | -/- | 21/43 | -/- (-) |
| d-0c-06 | -/- | -/- | -/- | -/- | 778/116 | -/- | 20/39 | -/- (-) |
| d-0c-07 | 1736/- | -/- | 41/- | -/- | 1331/1011 | -/- | 10/9 | -/- (-) |
| d-1b-00 | -/- | -/- | -/- | -/- | 205/122 | 205/122 | 4/9 | 4/9 (1/8/0) |
| d-1b-02 | -/- | -/- | -/- | -/- | 360/122 | -/122 | 8/9 | -/9 (1/8/0) |
| d-29-02 | -/- | -/- | -/- | -/- | 16/1083 | 16/1083 | 1/8 | 1/8 (1/1/6) |
| d-31-04 | 35/88 | 35/88 | 6/2 | 6/2 | 135/2 | 0/0 | 2/2 | 0/0 (0/0/0) |
| d-32-03 | -/- | -/- | -/- | -/- | 375/209 | 375/209 | 5/5 | 5/5 (3/2/0) |
| d-48-00 | -/1 | -/0 | -/1 | -/0 | 28/1321 | 0/0 | 1/9 | 0/0 (0/0/0) |
| d-5b-00 | 5/6 | -/6 | 1/6 | -/6 | 28/46 | -/46 | 2/4 | -/4 (1/3/0) |
| d-6e-04 | -/- | -/- | -/- | -/- | -/1 | -/1 | -/1 | -/1 (0/1/0) |
| d-83-00 | 1/- | 1/- | 1/- | 1/- | 158/2332 | 158/2332 | 1/14 | 1/14 (1/2/11) |
| d-90-00 | 125/181 | 0/0 | 3/4 | 0/0 | 88/134 | 18/2 | 2/5 | 1/2 (0/2/0) |
| d-90-04 | 182/94 | 0/0 | 1/3 | 0/0 | 272/75 | 221/5 | 7/24 | 3/5 (0/5/0) |
| d-98-04 | -/- | -/- | -/- | -/- | 13/62 | 13/62 | 2/3 | 2/3 (1/2/0) |
| d-bf-03 | -/- | -/- | -/- | -/- | 39/59 | 39/59 | 3/4 | 3/4 (1/3/0) |
| d-bf-04 | -/- | -/- | -/- | -/- | 80/78 | 80/78 | 6/8 | 6/8 (1/7/0) |
| d-bf-05 | -/- | -/- | -/- | -/- | 159/202 | 159/202 | 5/5 | 5/5 (3/2/0) |
| d-c5-03 | 611/995 | 611/995 | 9/21 | 9/21 | 209/191 | 209/191 | 3/7 | 3/7 (3/4/0) |
| d-c9-00 | -/- | -/- | -/- | -/- | -/1 | -/0 | -/1 | -/0 (0/0/0) |
| d-c9-01 | 8/39 | 0/0 | 3/39 | 0/0 | -/- | -/- | -/- | -/- (-) |
| d-ca-03 | -/2 | -/2 | -/2 | -/2 | -/- | -/- | -/- | -/- (-) |
| d-ca-02 | -/- | -/- | -/- | -/- | -/10 | -/10 | -/10 | -/10 (0/10/0) |
| d-d1-01 | 72/13 | 72/13 | 3/13 | 3/13 | 94/173 | 94/173 | 3/5 | 3/5 (3/2/0) |
| d-d5-00 | 213/114 | 213/114 | 1/19 | 1/19 | 87/253 | 87/246 | 3/16 | 3/15 (5/10/0) |
| d-d6-00 | -/- | -/- | -/- | -/- | -/1 | -/1 | -/1 | -/1 (0/1/0) |
| d-e7-00 | -/- | -/- | -/- | -/- | 88/68 | 0/1 | 2/1 | 0/1 (0/1/0) |
| d-e7-02 | -/- | -/- | -/- | -/- | 130/67 | -/0 | 1/1 | -/0 (0/0/0) |
| d-e9-00 | 7/1 | 7/1 | 1/1 | 1/1 | 103/74 | 103/74 | 3/12 | 3/12 (1/11/0) |
| d-f2-02 | 71/97 | 71/97 | 5/9 | 5/9 | 54/65 | 54/65 | 2/3 | 2/3 (1/2/0) |

## 5.9 Related Work

We already discussed some related work in Section 5.1 and Section 5.2, focusing on pros and cons of different *search strategies* in existing program repair approaches. In this section, we focus the discussions on a few closely related topics, and refer interested readers to a comprehensive survey for automatic program repair in general [113].

**Generating fixing ingredients.** Generating fixing ingredients is a critical step for automatic program repair approaches as it ultimately determines whether the generated search space contains a (correct) patch. To increase the likelihood of including the right fixing ingredients into the search space, researchers have proposed various edit templates. For example, GenProg uses three generic edit templates that append, replace and delete existing *statements* from the buggy file(s) under repair [162]. CapGen improves these three edit templates to consider edits at a more fine-grained expression level [163]. SearchRepair, in contrast, generalizes the idea of taking existing code fragments to a more coarse-grained level to include larger code fragments from an external code bank [67]. MutRepair takes inspiration from mutation testing research to randomly mutate existing *expressions* using standard generic mutation operators [28].

Fixing ingredients that target more specific fault classes have also been explored. For example, PAR generates fixing ingredients based on repair templates that are manually mined from developer patches [71]. ELIXIR targets method invocations by aggressively synthesizing method calls as fixing ingredients [138]. Tan et al. [150] propose anti-patterns to exclude seemingly trivial repairs. Most existing semantics-based approaches target `if` conditions and the right hand side of assignments via program synthesis [86, 95, 97, 107, 116, 169].

As discussed in Section 5.4, we consider 8 generic edit templates as a proof of concept. The novelty of this work lies in the effective and efficient search strategy, but our approach is orthogonal to recent research progress of generating diverse fixing ingredients, as long as they can be encoded as program variations via conditional statements or expressions.

**Patch quality.** Recent studies reveal that automatically generated patches tend to overfit to the tests used for generating repair [86, 113, 145]. To that end, researchers have explored different ways of measuring patch quality to tackle the weak specification of existing tests. The most widely adopted approach is to manually compare automatically generated patches against the developer match, using the developer patch as a gold standard [58, 95, 107, 139, 163]. However, manually checking syntactic, semantic, or functional equivalence of two programs is time-consuming, hard to scale, and even subjective. A more automated and objective approach is to use a high-quality held-out test suite to evaluate how likely the automatically generated patches can generalize to other tests [86, 114, 145].

In this work, we extend existing best practices of measuring patch quality with our symbolic-execution-based program verifier. Learning from prior work, we carefully distinguish different kinds of patches (i.e., plausible, generalizable, correct) in the evaluation.

**Multi-edit patches.** Generating multi-edit patches remains an open challenge in program repair research due to combinatorial explosion of fixing ingredients, and yet empirical evidence

suggests that a nontrivial portion of developer patches require making several edits [61, 175]. Most existing approaches can in theory generate multi-edit patches. For example, GenProg incrementally adds more edits to the pool of patch candidates as part of the genetic programming search [90, 162]. S3 can generate multi-edit patches by synthesizing patches at multiple suspicious locations separately, and then consolidating them into a single patch [86]. However, most existing approaches are not effective at generating multi-edit patches in practice, except for two approaches that specifically target this problem. Angelix can target multiple suspicious expressions at the same time by using symbolic execution to capture inter-location dependencies as constraints, but often require multiple suspicious locations to be close for symbolic execution to be effective [107]. Hercules exploits the fact that similar code changes are often made to similar locations (termed sibling relationship) to pro-actively derive similar edits at multiple locations at the same time [139].

Our approach can effectively find multi-edit patches because of the systematic search enabled by variational execution. Compared to Angelix and Hercules, our approach can generate multi-edit patches that are more general, independent of whether fixing locations are close or whether the fixing ingredients are similar. The systematic search of variational execution can often effectively identify multi-edit patches if they exist in the search space.

**Patch ranking.** Existing program repair approaches often have an internal component that ranks *patch candidates*, prioritizing those that are more likely to pass all tests, or have higher quality [26, 86, 90, 95, 97, 106, 107, 162, 163]. To rank *patch candidates*, existing approaches exploit different information, such as the number of passing and failing tests [90, 162], syntactic and semantic distance to the original program [26, 86, 106, 107], and probabilistic models learnt from existing human patches [87, 97, 163, 169].

In this work, patch ranking is not part of the search, but rather a post-processing step to prioritize presenting high-quality patches to developers. We rank *plausible patches* instead of *patch candidates*, thus ranking based on the number of passing and failing tests does not apply to our problem. However, orthogonal to other ranking strategies, we use dynamic runtime information for ranking and thus our approach can potentially complement existing ranking strategies that use syntactic and semantic distance or probabilistic models. Similar to our work, JAID also finds multiple plausible patches and rank them, but based on fault localization suspiciousness [23]. The work of Xiong et al. [168] on classifying correct patches is conceptually closest to our ranking in that they also compare execution traces. Despite that they solve a slightly different problem (i.e., patch classification rather than patch ranking), their ideas inspire our ranking strategies.

## 5.10 Summary

Existing approaches to automatic program repair essentially solve a search problem in which a trade-off needs to be made between *edit expressiveness* and *search effectiveness*. While most existing work sacrifice one or another, our work presented in this chapter strikes a balance between the two aspects, by using variational execution to effectively search a large search space of diverse fixing ingredients. Similar to Chapter 4, we first created a meta-program that contains

abundant fixing ingredients encoded as `if` conditionals. Next, we used variational execution to collect test results by running the meta-program against a test suite. Finally, using the test results that are encoded as Boolean constraints, we found patches by solving a well-studied satisfiability problem. A thorough evaluation on IntroClassJava and Defects4J reveals that our approach is effective at repairing programs of varying sizes by generating many high-quality and multi-edit patches.

This chapter demonstrates yet another application of variational execution, in addition to higher-order mutation testing as described in Chapter 4, showing that variational execution is a viable way of exploring large search spaces and uncovering interesting interactions.

Although the overall recipe of using variational execution is the same in Chapter 4 and Chapter 5, the techniques are different from the engineering perspective. The search space of automatic program repair is more challenging because statement-level changes as speculative variations are common. Statement-level changes affect both meta-program generation and variational execution: We rebuilt meta-program generation in order to use fault localization to guide edit generation, and more importantly, encode statement changes as `if` conditionals, extracting them into separate methods via invasive refactoring if necessary. We made several adaptations (e.g., fast mode, bounded search) to variational execution to cope with statement-level changes, because unlike expression changes in mutation testing, statements can easily disrupt control flow, posing more challenges to the already expensive handling of control transfer (Section 3.4), in particular exception handling (Section 3.5), and potentially causing more infinite loops that can heavily impact performance of variational execution. Most of these issues can be addressed with careful engineering, as we have demonstrated by evaluating our approach on two of the largest subjects in Defects4J. While the conceptual idea of using variational execution is more beneficial to the research community, we hope that the ways we make engineering tradeoffs as presented in this chapter can inspire new ways of applying other similar techniques, such as symbolic execution and model checking.

# Chapter 6

# Conclusions

Variations are ubiquitous and can take two forms: *intentional variations* that are manually introduced to tweak program behavior and *speculative variations* that are automatically generated to analyze the program. While variations are useful, they can often yield a large configuration space or search space that challenges existing software analysis techniques.

Variational execution has been proposed to systematically analyze vast configuration spaces of *intentional variations*, indicating that the technique has the potential to explore large search spaces of *speculative variations*. In Chapter 2, we analyzed existing successful applications of variational execution to derive key criteria that can be used to gauge the potential of future applications. Using these criteria, we discovered that, indeed, a wide range of software engineering problems that heavily use speculative variations can benefit from variational execution, such as higher-order mutation testing and automatic program repair.

Since existing implementations of variational execution fail to handle the large scale of speculative variations, in Chapter 3, we built a new variational execution engine based on bytecode transformation, focusing on efficiency, scalability, and extensibility. We compared our new implementation with the state of the art, showing that our approach significantly outperforms existing work on all subjects. We call our new implementation VarexC and use it consistently throughout this thesis to explore speculative variations.

In Chapter 4, we applied variational execution to search for strongly subsuming higher-order mutants (SSHOM), a specific form of higher-order mutant that denotes subtle fault. Our approach transforms the search for SSHOMs into a satisfiability problem: First, we used variational execution to run all first-order mutants (and their combinations) against a test suite, representing test results compactly as Boolean constraints. Next, we used the test results (i.e., Boolean constraints) to construct a propositional formula that bakes in the definition of SSHOM. Finally, we used off-the-shelf SAT solving techniques to get all solutions to the propositional formula, each of which corresponds directly to a SSHOM. Using this approach, not only can we find SSHOMs much faster, but also we can identify all SSHOMs in the given search space. Since our approach explores the search space systematically, we had the unique opportunity to study the characteristics of the search space. Using the identified characteristics, we further designed a prioritized search that is lightweight, but effective at finding many SSHOMs in large systems.

Using a similar recipe in Chapter 5, we applied variational execution to automatic program repair. The way we encode the search as a satisfiability problem is conceptually similar to

higher-order mutation testing, but the techniques are different from the engineering perspective because statement-level changes can be challenging for both meta-program generation and variational execution. Comparing to existing work, our approach makes a better tradeoff between edit expressiveness and search effectiveness, and this tradeoff further sheds light on several open challenges, such as patch quality and multi-edit patches. We evaluated our approach on two widely used datasets, showing that the new search is effective in multiple aspects: fixing many bugs, generating many high-quality patches, identifying many multi-edit patches, and distinguishing patch quality based on dynamic information.

**Thesis Statement.** We conclude that this thesis provides enough evidence that validates the thesis statement set forth in Chapter 1, where we hypothesized that research problems of speculative variations can benefit from variational execution. To validate this hypothesis, we started with a conceptual analysis of applicability, using three criteria derived from existing successful applications to identify two promising problems—higher-order mutation testing and automatic program repair (Chapter 2). With an improved implementation of variational execution (Chapter 3), we showed that variational execution can indeed be useful for analyzing speculative variations. For higher-order mutation testing, we were the first to perform a complete search of SSHOMs for medium-sized programs (Chapter 4). Furthermore, we demonstrated that the complete search enabled by variational execution can facilitate a systematic study of SSHOM characteristics, which can be useful for designing new heuristics-based search strategies that scale to large real-world programs. For automatic program repair, we demonstrated that a systematic exploration of the search space is effective at finding (many) patches (Chapter 5). In contrast to existing work, our approach is also useful for finding high-quality and multi-edit patches within a large search space. To take full advantage of variational execution, we further used the dynamic information obtained from variational execution runtime to distinguish high-quality patches from those that merely pass all provided tests.

## 6.1 Future Work: Variational Execution

This thesis is devoted to improving variational execution and applying it to important problems of speculative variations. Reflecting upon our experience, we highlight several future directions.

### 6.1.1 Improving Variational Execution

We made significant improvement to variational execution, making it more efficient, more scalable, and more extensible. These improvements have direct benefits to variational execution, but our implementation (VarexC) is far from being a full-fledged tool that can be applied to analyze arbitrarily complex programs.

**Environment Barrier.** The main inhibitor for broad adoption is the environment barrier between code that can be executed with variational execution and code that needs to be executed with native runtime (e.g., due to security reasons or use of native methods). Other similar techniques such as symbolic execution face a similar challenge. For example, despite more

than 10 years of active development involving more than 70 contributors,[1] KLEE remains a research instrument that faces challenges from analyzing library code. In this thesis, we manually implemented model classes that were needed for running our experiment subjects, but ideally, a mature variational execution engine should provide a much wider range of model classes to support advanced language features like reflection and lambda ( Section 3.5).

Based on our experience, full support for the JDK classes could mitigate a lot of the environment barrier issues, as they are commonly used in application code and often serve as the basis for third-party libraries. Ideally, a full-fledged implementation of variational execution should provide necessary infrastructure for (1) identifying Java classes that require model classes, and more importantly, (2) automatically generating model classes. In this thesis, both steps are performed manually, but they are automatable based on our experience. For example, to identify classes that require model classes, we can use simple heuristics, such as looking for the `native` keyword, which indicates the use of native methods. Automatically generating model classes is viable in most of the cases we observed, especially in cases where non-variational code (i.e., code that needs to be executed without variational execution) has no side effect to the program state (e.g., delegating expensive mathematical computation to native libraries written in C). For these cases, we can simply execute the non-variational code repeatedly, once for each combination of concrete values involved, and finally compress the returned values into a conditional value again for the rest of variational execution (Section 3.3.2). Note however that there could be better ways of implementing model classes than repeatedly executing non-variational code, for example, by using specialized data structures and custom access patterns (Section 3.5). But automatically generated model classes can serve as a reliable baseline and more efficient implementations can be incrementally designed if needed for performance reasons.

More broadly, techniques for addressing environment barrier issues in variational execution can potentially be broadened to tackle similar challenges in closely related techniques, such as symbolic execution and model checking. For example, the core interpreter of Java Pathfinder has been adapted to perform model checking [49], symbolic execution [4], and variational execution [110], but they face a similar issue with regard to native methods. The solution is also similar—to implement model classes that model the expected behavior in model checking, symbolic execution, or variational execution. We were the first to implement variational execution via bytecode transformation, and there is no similar bytecode implementation for symbolic execution or model checking at the time of writing. However, we suspect that most of the challenges are similar and solutions are transferrable, in particular the techniques for handling environment barrier issues.

**Performance.** While the three key criteria discussed in Chapter 2 are useful for *conceptually* evaluating the potential of applying variational execution, actual performance of running variational execution could be another main inhibitor that hinders adoption *in practice*. We argue that performance of variational execution is an empirical question. On the one hand, for intentional variations that are often carefully introduced to have manageable interactions, empirical studies have shown that the complexity of how variations interact is often low [110, 134], making a complete exploration of the configuration space feasible. On the other hand,

[1] https://github.com/klee/klee

for speculative variations that are often generated randomly, the performance of variational execution largely depends on how those variations interact at runtime for the specific tests used. For example, if interactions of variations do not drastically modify program state, a lot of sharing can be exploited by variational execution to maximize performance. That said, we did observe a few rare cases where interactions are heavy, for both higher-order mutation testing (Chapter 4) and automatic program repair (Chapter 5). We argue that those corner cases result from *essential complexity* of the complex search space, and we can often side step those corner cases by carefully restricting the search space.

To ease adoption, it would be nice to have some means of predicting whether running variational execution on a specific program with provided tests would incur significant performance overhead. As discussed above, performance largely depends on the specific variations and tests, but heuristics can be used. For example, one could statically analyze the locations of variations. Clustered variations that heavily modify a handful of variables that have a infinite domain are more likely to cause expensive interactions. Another way of approximating performance is to look for certain expensive operations. In the case of VarexC, array operations that cross the environment barrier are noticeably slow (Section 3.5) because transformation between a variational array and the concrete arrays it represents could be expensive in the current implementation (Chapter 3). Several heuristics should be combined for better prediction. For example, sparse variations that modify different variables in different methods might also cause heavy interactions if they influence the program state via implicit control flow and data flow.

Based on our experience, the performance of variational execution could be further improved by performing several optimizations. First, we suspect that overall performance can be improved drastically by having a more efficient way of representing and manipulating binary decision diagrams (BDD), which are heavily used in variational execution to represent partial configuration spaces. We have briefly explored multi-terminal BDD as an alternative way of representing conditional values (not included in this dissertation). At the time of writing this dissertation, we are still working on a reliable implementation, but preliminary results indicate that there is room for improvement in the way we use BDD. More research in this direction can also benefit other important domains that use BDDs, such as symbolic execution, model checking, and hardware design.

Second, as discussed in Section 3.5, more specific optimizations can be performed if necessary by designing specialized data structures that maximize performance of certain operations [112, 158], such as a `List` data structure that optimizes for appending elements rather than lookup. Common access patterns can also be exploited to speedup common operations, as demonstrated in the work of Lazarek [84] where list iteration via `iterator` can be optimized by using a specialized model class implementation.

Finally, we can use heuristics to restrict running variational execution in a smaller search space, and incrementally tackle the overall large search space in a divide-and-conquer manner. We have briefly explored several ways of restricting the search space of automatic program repair (Chapter 5), such as a bounded search that limits the interaction degree and a fast mode that focuses on exception-free paths. Note however that, running variational execution separately for partial search spaces could miss potential interactions *across* partial search spaces. Hence, this style of divide-and-conquer exploration might miss interactions as opposed to a more heavyweight but complete exploration of the whole search space, unless for cases where

interactions are statically determined to be impossible [134]. However, restricting the search space can still be beneficial if combined with domain knowledge of the problem. For example, in the case of program repair, if we know beforehand that modifying a single method is sufficient, we could group variations based on method boundaries and search in those smaller spaces.

More broadly speaking, performance improvement to variational execution is likely transferrable to similar techniques such as symbolic execution and model checking because they can often be implemented similarly. For example, our experience of implementing variational execution via bytecode transformation rather than modifying a language interpreter can potentially inform a more scalable implementation of symbolic execution, because the state-of-the-art symbolic execution for Java is based on modifying Java PathFinder, similar to how variational execution was alternatively implemented in VarexJ [110]. Moreover, researchers have also tried ways to limit symbolic execution to partial code [107] or partial inputs [42] in order to reduce performance overhead, in a similar spirit as the way we restrict the search space of variational execution.

### 6.1.2 New Applications

By analyzing existing work on variational execution, we derive three key criteria that predict a successful application of variational execution. This strategy of deriving application criteria turns out to be useful, as demonstrated with the two successful applications of variational execution. We hope that the research community can benefit from these criteria when gauging future applications of variational execution, such as search problems that are similar to higher-order mutation testing and automatic program repair, many of which fall under the umbrella of search-based software engineering [47].

There could be different strategies of applying variational execution to solve new problems. Traditionally for intentional variations, researchers had success with performing a *complete* exploration of the search space on real-world large systems such as WordPress [117], which is often feasible based on recent empirical findings that variations do not interact heavily altogether. For cases where interactions could be expensive, this thesis demonstrates two alternative ways of applying variational execution.

First, for higher-order mutation testing in Chapter 4, we still performed a complete search considering all variations in the search space, but only for small to medium-sized programs. Because of the complete exploration, we could identify *all* SSHOMs for our subject programs and later empirically study what they look like, with the goal to improve existing heuristics-based search strategies or inform new design (Section 4.5). Taking advantage of the complete search enabled by variational execution, we can derive insights with more confidence than learning from an incomplete and potentially biased sample of SSHOMs. This way of using variational execution assumes that insights derived from small to medium-sized programs are transferrable to larger programs or other types of program in general. We argue that this assumption is reasonable if we focus primarily on insights that are largely independent of specific program structure, as discussed in Section 4.4.

Second, for automatic program repair in Chapter 5, we perform the search for large real-world programs (e.g., Apache Math and Closure), but used different strategies to restrict the search space for performance reasons. As discussed above, performance can potentially be improved

if more research is devoted and thus the restrictions on the search space can potentially be relaxed in the future. But even with limited performance, we can use domain knowledge to help us narrow the search. For example, in contrast to higher mutation testing where high-degree SSHOMs are potentially valuable, we prefer low-degree patches in automatic program repair because small fixes that concisely fix the buggy behavior are easy to safeguard manually and can reduce the likelihood of introducing regression errors. Based on this domain knowledge, a bounded search that limits the number or edits involved in a patch is not only useful for alleviating performance issues but also favorable in practice.

There could be other ways of using variational execution, such as performing variational execution on the most relevant part of the code, similar to how symbolic execution is selectively applied in Angelix [107]. While we can learn from existing ways of using symbolic execution, we suspect that the opposite direction of transferring knowledge is viable. That is, the methodology of applying variational execution can potentially inform novel ways of using symbolic execution and model checking.

## 6.2 Future Work: Higher-Order Mutation Testing

Higher-order mutants are not commonly used in practice mostly because valuable ones such as SSHOMs are difficult to find and thus little is known about them. Our work advances this research direction by proposing two new search strategies that can effectively find many SSHOMs for a given test suite. This way, our work can facilitate future studies on SSHOMs.

One possible future direction is to use our techniques as research instrument for creating bugs that are hard to detect by the given tests. These bugs can then be used to evaluate other techniques, such as fault localization and automatic program repair as discussed in Chapter 4. This way, we can easily generate a large number of challenging bugs to perform a objective and comprehensive evaluation of other research, rather than relying solely on a few manually curated datasets that are used heavily to the extent of raising overfitting concerns [33, 154].

Another potential direction is to study how to generate SSHOMs without first executing all constituent first-order mutants against the given test suite. Simple first-order mutants are easier to generate than SSHOMs because they can be created randomly, independent of any tests. In contrast, whether a higher-order mutant is SSHOM depends on test results of its constituent first-order mutants (Chapter 4). For this reason, all existing approaches to finding SSHOMs, including ours, need to execute many first-order mutants against the given tests before actually generating SSHOMs. Moreover, the set of SSHOMs is likely different after changing the source code or test suite. To make SSHOMs more useful in practical mutation testing scenarios, we need reliable ways of generating or approximating SSHOMs statically without running first-order mutants. Although not for SSHOMs, Just et al. [64] showed that static information such as program context can be used to approximate dominator mutants, a special kind of mutants that are similarly defined in terms of test results of first-order mutants. Furthermore, future research can use our techniques to generate abundant SSHOMs for a more systematic study of their characteristics to inform static generation.

Finally, our techniques can be generalized to other types of higher-order mutants. Jia and Harman [56] outlined a taxonomy of higher-order mutants based on subsuming relation of

first-order mutants, but only focused on SSHOMs. Other types of higher-order mutants or different taxonomies might also be useful, and can potentially benefit from our techniques, for example, by tweaking the encoding of satisfiability problem (Section 4.3.3).

## 6.3 Future Work: Automatic Program Repair

Repair tools typically contain two important components—fixing ingredient generation and search strategy. This work only focuses on improving the search strategy, by reusing existing infrastructure of classic GenProg for conceptual simplicity. Our work is orthogonal to recent research progress in fixing ingredient generation. To fully realize the potential of our work, more research is needed to incorporate recent progress of fixing ingredient generation and improve the search strategy.

**Fixing Ingredients Generation.** The pool of fixing ingredients ultimately decides whether a patch can be found. In Chapter 5, we showed that fixing ingredients generated from old-fashion edit templates and a simplistic fault localization technique can already yield many high-quality patches. It is thus likely that more bugs can be fixed and more high-quality patches can be identified if we incorporate more recent advanced edit templates and fault localization techniques. That said, adding more and more fixing ingredients all over the code base might pose severe challenges to the scalability of variational execution tools. To mitigate this issue, we can use heuristics to be selective about fixing ingredients. For example, future work could intelligently pick promising edit templates based on code structure, bug fixes in the past, or developer insights. This way, we can focus the search on a highly promising search space rather than a unnecessarily large one. Moreover, fixing ingredients can be generated in batches based on fault localization insights. For example, we could learn from Hercules [139] to generate fixing ingredients for other code locations that are similar syntactically or semantically, as opposed to all locations that the fault localization technique deems even slightly suspicious.

**Search Strategy.** At the core of a repair tool is the search strategy. Researchers have proposed different strategies in the past, but they either search systematically in a confined search space or forgo a systematic search in exchange for more expressive power in the search space (Section 5.1). Our search strategy diverges drastically from existing trends in that we perform a systematic search in a large expressive search space, by relying on variational execution to aggressively exploit the similarities among test executions. This approach has both advantages and disadvantages.

On the bright side, as our evaluation shows in Chapter 5, our approach can effectively and efficiently identify (many) patches even for large programs like Apache Math and Closure. Conceptually, a systematic search like ours can identify *all* plausible patches if they are within the given search space. Finding all plausible patches has several benefits. First, if a patch is not found with our approach, we can be certain that the search space lacks necessary fixing ingredients and so we can move on to generate more promising ingredients. In contrast, heuristics-based search strategies cannot make such claims and so time could be wasted searching in a hopeless search space. Second, since we find all plausible patches, we also find the high-quality ones if

they exist. As shown empirical in our work (Chapter 5) and prior work [96], cherry picking the high-quality patches from a much larger pool of plausible patches is challenging. However, this issue is not relevant to our approach because we do not cherry pick, but identify all patches. Finally, our approach stands out from existing work because the systematic search powered by variational execution can efficiently explore interactions of fixing ingredients to identify *all* multi-edit patches. Most heuristics-based search strategies can rarely find multi-edit patches because the search space becomes intractable quickly if multiple fixing ingredients are combined.

However, there are two potential issues with our approach. First of all, although theoretically feasible, our systematic search in practice can only go as far as the variational execution engine scales. We can find all patches if our variational execution tool can finish exploring the search space in a reasonable time. But on the other hand, we cannot find any patches if variational execution does not terminate within the given time budget, whereas heuristics-based search might find some patches over time. We hope that this issue can be mitigated with performance improvement to variational execution, for example, by exploring the ideas outlined in Section 6.1. Moreover, we can often circumvent the performance bottleneck by making assumptions to confine the search space, as discussed in Section 5.5. Finally, we argue that variational execution is a recent and effective way of performing a systematic search, but not the only way. Future work can swap out the variational execution component if more efficient techniques emerge in the future. The second potential issue of our approach is that finding all patches might overwhelm developers. We proposed simple patch ranking mechanisms that work well for our evaluation subjects, but more research opportunities remain to automatically distinguish high-quality patches from low-quality ones, for example, by leveraging more static or dynamic information. That said, we argue that the issue of patch ranking is not exclusive to our work, but rather a common challenge for all repair techniques. This issue becomes more obvious in our work than most existing techniques because they are often satisfied with the first plausible patch.

Independent of our technical approach, we hope that our work can inform the design of new search strategies. Mirroring our workflow in higher-order mutation testing, future work could systematically study characteristics of high-quality patches and multi-edit patches, and then design new lightweight heuristics-based search strategies that exploit those identified characteristics. For example, although not explicitly discussed in this work, since our patch ranking results indicate that dynamic runtime information can be useful for predicting patch quality, future search strategies can potentially exploit this finding to prune the search space in the early stage and focus on patch candidates that have little influence to passing tests but moderate effect on failing tests.

# Bibliography


[1] 2018. Tricentis Software Fail Watch Finds 3.6 Billion People Affected and $1.7 Trillion Revenue Lost by Software Failures Last Year. (Jan. 2018). [ Cited on page 75. ]

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-Based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*. IEEE Computer Society, USA, 89–98. [ Cited on page 52. ]

[3] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/170035.170072 [ Cited on page 65. ]

[4] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2007. JPF–SE: A Symbolic Execution Extension to Java PathFinder. In *Tools and Algorithms for the Construction and Analysis of Systems*, Orna Grumberg and Michael Huth (Eds.). Vol. 4424. Springer Berlin Heidelberg, Berlin, Heidelberg, 134–138. https://doi.org/10.1007/978-3-540-71209-1_12 [ Cited on page 119. ]

[5] 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 [ Cited on page 8. ]

[6] 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 [ Cited on page 12. ]

[7] 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 [ Cited on pages 2, 12, 13, 17, 41, 42, 56, and 57. ]

[8] 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 [ Cited on pages 13, 17, 41, and 77. ]

[9] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proceedings of the ACM on Programming Languages* 3,

OOPSLA (Oct. 2019), 159:1–159:27. https://doi.org/10.1145/3360585 [ Cited on page 75. ]

[10] 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 [ Cited on page 11. ]

[11] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, Hong Kong, China, 306–317. https://doi.org/10.1145/2635868.2635898 [ Cited on page 82. ]

[12] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. 2012. Secure Multi-Execution through Static Program Transformation. In *Formal Techniques for Distributed Systems*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Holger Giese, and Grigore Rosu (Eds.). Vol. 7273. Springer Berlin Heidelberg, Berlin, Heidelberg, 186–202. https://doi.org/10.1007/978-3-642-30793-5_12 [ Cited on page 42. ]

[13] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*. ACM Press, Portland, Oregon, USA, 83–101. https://doi.org/10.1145/2660193.2660212 [ Cited on page 42. ]

[14] Jonathan Bell and Luís Pina. 2018. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. (2018), 31 pages. https://doi.org/10.4230/LIPICS.ECOOP.2018.17 [ Cited on page 43. ]

[15] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and W. Rance Cleaveland (Eds.), Vol. 1579. Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207. https://doi.org/10.1007/3-540-49059-0_14 [ Cited on page 57. ]

[16] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL$^{\text{LIFT}}$: Statically Analyzing Software Product Lines in Minutes Instead of Years. *ACM SIGPLAN Notices* 48, 6 (June 2013), 355–364. https://doi.org/10.1145/2499370.2491976 [ Cited on page 43. ]

[17] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–26. https://doi.org/10.1145/3276519 [ Cited on page 57. ]

[18] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 168–178. https://doi.org/10.1145/2025113.2025139 [ Cited on page 42. ]

[19] Randal E. 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 [ Cited on page 58. ]

[20] 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. [ Cited on page 2. ]

[21] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Comput. Netw.* 41, 1 (Jan. 2003), 115–141. https://doi.org/10.1016/S1389-1286(02)00352-3 [ Cited on pages 1 and 12. ]

[22] 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 Computer Society, Miami Beach, Florida, USA, 463–475. https://doi.org/10.1109/ACSAC.2007.37 [ Cited on page 12. ]

[23] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-Based Program Repair without the Contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 637–647. [ Cited on pages 91, 93, 95, and 115. ]

[24] 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 [ Cited on page 2. ]

[25] 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 [ Cited on pages 14, 22, 42, and 57. ]

[26] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Vol. 9780. Springer International Publishing, Cham, 383–401. https://doi.org/10.1007/978-3-319-41540-6_21 [ Cited on page 115. ]

[27] 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 [ Cited on pages 11 and 42. ]

[28] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Verification and Validation 2010 Third International Conference on Software Testing*. IEEE Computer Society, Paris, France, 65–74. https://doi.org/10.1109/ICST.2010.66 [ Cited on page 114. ]

[29] 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 [ Cited on pages 11 and 42. ]

[30] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-Based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 655–666. https://doi.org/10.1145/2884781.2884821 [ Cited on pages 72 and 83. ]

[31] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. 2017. Efficient Family-Based Model Checking via Variability Abstractions. *International Journal on Software Tools for Technology Transfer (STTT)* 19, 5 (Oct. 2017), 585–603. https://doi.org/10.1007/s10009-016-0425-2 [ Cited on page 43. ]

[32] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. 2019. A Systematic Mapping Study on Higher Order Mutation Testing. *Journal of Systems and Software* 154 (Aug. 2019), 92–109. https://doi.org/10.1016/j.jss.2019.04.031 [ Cited on page 53. ]

[33] 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. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*. ACM Press, Tallinn, Estonia, 302–313. https://doi.org/10.1145/3338906.3338911 [ Cited on pages 53, 90, 91, 92, 94, 95, and 122. ]

[34] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test - AST '16*. ACM Press, Austin, Texas, 85–91. https://doi.org/10.1145/2896921.2896931 [ Cited on pages 76 and 80. ]

[35] Thomas Durieux and Martin Monperrus. 2016. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. (Feb. 2016), 7. [ Cited on page 91. ]

[36] Emelie Engström and Per Runeson. 2011. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology* 53, 1 (Jan. 2011), 2–13. https://doi.org/10.1016/j.infsof.2010.05.011 [ Cited on page 43. ]

[37] 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 [ Cited on page 8. ]

[38] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*. Västeras, Sweden, 313–324. https://doi.org/10.1145/2642937.2642982 [ Cited on page 109. ]

[39] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033 [ Cited on page 36. ]

[40] Ahmed S. Ghiduk. 2016. Reducing the Number of Higher-Order Mutants with the Aid of Data Flow. *e-Informatica Vol. X* (2016), 2016; ISSN 18977979. https://doi.org/10.5277/

E-INF160102 [ Cited on page 52. ]

[41] Ahmed S. Ghiduk, Moheb R. Girgis, and Marwa H. Shehata. 2017. Higher Order Mutation Testing: A Systematic Literature Review. *Computer Science Review* 25 (Aug. 2017), 29–48. https://doi.org/10.1016/j.cosrev.2017.06.001 [ Cited on pages 52 and 53. ]

[42] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036 [ Cited on page 121. ]

[43] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2017. The Theory of Composite Faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Tokyo, Japan, 47–57. https://doi.org/10.1109/ICST.2017.12 [ Cited on pages 52 and 53. ]

[44] Mark Harman, Yue Jia, and William B. Langdon. 2010. A Manifesto for Higher Order Mutation Testing. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, Paris, France, 80–89. https://doi.org/10.1109/ICSTW.2010.13 [ Cited on page 53. ]

[45] Mark Harman, Yue Jia, and William B. Langdon. 2011. Strong Higher Order Mutation-Based Test Data Generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 212–222. https://doi.org/10.1145/2025113.2025144 [ Cited on page 52. ]

[46] 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 [ Cited on pages 50, 52, 54, 59, 61, and 72. ]

[47] Mark Harman and Bryan F Jones. 2001. Search-Based Software Engineering. *Information and Software Technology* 43, 14 (Dec. 2001), 833–839. https://doi.org/10.1016/S0950-5849(01)00189-6 [ Cited on page 121. ]

[48] 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 [ Cited on pages 2 and 3. ]

[49] 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 [ Cited on pages 17 and 119. ]

[50] 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. [ Cited on pages 11 and 42. ]

[51] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. IEEE Computer Society Press, Washington, DC, USA, 191–200. [ Cited on page 52. ]

[52] Chihiro Iida and Shingo Takada. 2017. Reducing Mutants with Mutant Killable Precondition. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Tokyo, Japan, 128–133. https://doi.org/10.1109/ICSTW.2017.29 [ Cited on page 52. ]

[53] Yue Jia. 2013. *Higher Order Mutation Testing*. Ph.D. Dissertation. [ Cited on page 61. ]

[54] 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, China, 249–258. https://doi.org/10.1109/SCAM.2008.36 [ Cited on pages 50, 52, 53, 54, 61, 71, and 72. ]

[55] Yue Jia and Mark Harman. 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Testing: Academic Industrial Conference - Practice and Research Techniques (Taic Part 2008)*. IEEE, Windsor, UK, 94–98. https://doi.org/10.1109/TAIC-PART.2008.18 [ Cited on pages 71 and 72. ]

[56] 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 [ Cited on pages 1, 50, 52, 53, 54, 57, 58, 59, 61, 65, 71, 72, and 122. ]

[57] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62 [ Cited on pages 42 and 72. ]

[58] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871 [ Cited on pages 91, 92, 93, 95, and 114. ]

[59] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949 [ Cited on pages 52 and 53. ]

[60] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, San Jose, CA, USA, 315–326. https://doi.org/10.1145/2610384.2610388 [ Cited on page 72. ]

[61] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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 [ Cited on pages 52, 53, 81, 91, 92, 96, and 115. ]

[62] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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 [ Cited on pages 49, 52, and 53. ]

[63] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2011. Using Conditional Mutation to Increase the Efficiency of Mutation Analysis. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 50–56. https://doi.org/10.1145/1982595.1982606 [ Cited on page 56. ]

[64] 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 [ Cited on pages 70, 72, and 122. ]

[65] Christian Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *arXiv:1706.09357 [cs]* (June 2017). arXiv:1706.09357 [cs] [ Cited on page 31. ]

[66] 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 [ Cited on pages 2, 12, 41, 42, and 43. ]

[67] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Press, Lincoln, Nebraska, 295–306. https://doi.org/10.1109/ASE.2015.60 [ Cited on page 114. ]

[68] Christian Kern and Javier Esparza. 2010. Automatic Error Correction of Java Programs. In *Proceedings of the 15th International Conference on Formal Methods for Industrial Critical Systems (FMICS'10)*. Springer-Verlag, Berlin, Heidelberg, 67–81. [ Cited on page 83. ]

[69] 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 [ Cited on pages 12, 41, and 42. ]

[70] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo D'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 257–267. https://doi.org/10.1145/2491411.2491459 [ Cited on page 72. ]

[71] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 802–811. https://doi.org/10.1109/ICSE.2013.6606626 [ Cited on pages 76, 79, and 114. ]

[72] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252 [ Cited on page 57. ]

[73] K. N. King and A. Jefferson Offutt. 1991. A Fortran Language System for Mutation-Based Software Testing. *Software: Practice and Experience* 21, 7 (July 1991), 685–718. https://doi.org/10.1002/spe.4380210704 [ Cited on pages 52 and 56. ]

[74] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *2010 Asia Pacific Software Engineering Conference*. IEEE, Sydney, NSW, Australia, 300–309. https://doi.org/10.1109/APSEC.2010.42 [ Cited on pages 50 and 71. ]

[75] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2015. Employing Second-Order Mutation for Isolating First-Order Equivalent Mutants. *Software Testing, Verification and Reliability* 25, 5-7 (Aug. 2015), 508–535. https://doi.org/10.1002/stvr.1529 [ Cited on page 52. ]

[76] 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 [ Cited on pages 11 and 42. ]

[77] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutant Subsumption Graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, OH, USA, 176–185. https://doi.org/10.1109/ICSTW.2014.20 [ Cited on page 72. ]

[78] Bob Kurtz, Paul Ammann, and Jeff Offutt. 2015. Static Analysis of Mutant Subsumption. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Graz, Austria, 1–10. https://doi.org/10.1109/ICSTW.2015.7107454 [ Cited on page 72. ]

[79] 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 [ Cited on pages 63 and 72. ]

[80] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Chicago, IL, USA, 142–151. https://doi.org/10.1109/ICSTW.2016.41 [ Cited on page 72. ]

[81] Markus Kusano and Chao Wang. 2013. CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Silicon Valley, CA, USA, 722–725. https://doi.org/10.1109/ASE.2013.6693142 [ Cited on page 71. ]

[82] 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 [ Cited on pages 11, 12, 41, and 42. ]

[83] 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 [ Cited on pages 50, 53, 71, and 72. ]

[84] Lukas Lazarek. 2017. How to Efficiently Process $2^{100}$ List Variations. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 36–38. https://doi.org/10.1145/3135932.3135951 [ Cited on pages 33 and 120. ]

[85] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 429–432. https://doi.org/10.1145/2610384.2628052 [ Cited on page 71. ]

[86] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, Paderborn, Germany, 593–604. https://doi.org/10.1145/3106237.3106309 [ Cited on pages 76, 77, 80, 86, 90, 91, 93, 94, 97, 109, 114, and 115. ]

[87] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, 213–224. https://doi.org/10.1109/SANER.2016.76 [ Cited on page 115. ]

[88] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Zurich, Switzerland, 3–13. [ Cited on page 53. ]

[89] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software Quality Journal* 21, 3 (Sept. 2013), 421–443. https://doi.org/10.1007/s11219-013-9208-0 [ Cited on page 52. ]

[90] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104 [ Cited on pages 42, 79, 80, 86, and 115. ]

[91] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162 [ Cited on page 90. ]

[92] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 81–91. https://doi.org/10.1145/2491411.2491437 [ Cited on page 43. ]

[93] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java Virtual*

*Machine Specification Java SE 8 Edition.* [ Cited on page 20. ]

[94] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical Model-Based Bug Localization. *ACM SIGSOFT Software Engineering Notes* 30, 5 (Sept. 2005), 286. https://doi.org/10.1145/1095430.1081753 [ Cited on page 52. ]

[95] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 166–178. https://doi.org/10.1145/2786805.2786811 [ Cited on pages 75, 76, 79, 80, 114, and 115. ]

[96] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 702–713. https://doi.org/10.1145/2884781.2884872 [ Cited on pages 76, 79, 80, 81, 83, 86, 90, 93, 98, 99, and 124. ]

[97] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*. ACM Press, St. Petersburg, FL, USA, 298–312. https://doi.org/10.1145/2837614.2837617 [ Cited on pages 75, 76, 79, 80, 114, and 115. ]

[98] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Hangzhou, China, 468–478. https://doi.org/10.1109/SANER.2019.8667991 [ Cited on page 53. ]

[99] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz 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 [ Cited on pages 50, 52, and 71. ]

[100] Lech Madeyski and Norbert Radyk. 2010. Judy – a Mutation Testing Tool for Java. *IET Software* 4, 1 (2010), 32. https://doi.org/10.1049/iet-sen.2008.0038 [ Cited on page 56. ]

[101] Sonal Mahajan and William G.J. Halfond. 2014. Finding HTML Presentation Failures Using Image Comparison Techniques. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2642937.2642966 [ Cited on page 71. ]

[102] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE Press, Montreal, Quebec, Canada, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039 [ Cited on page 75. ]

[103] Matias Martinez and Martin Monperrus. 2019. Astor: Exploring the Design Space of Generate-and-Validate Program Repair beyond GenProg. *Journal of Systems and Software* 151 (May 2019), 65–80. https://doi.org/10.1016/j.jss.2019.01.069 arXiv:1802.03365 [ Cited on page 79. ]

[104] Pedro Reales Mateo, Macario Polo Usaola, and José Luis Fernández Alemán. 2013. Validat-

ing 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 [ Cited on pages 50, 52, and 71. ]

[105] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem Execution for Efficient Live Patch Testing. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. USENIX Association, Berkeley, CA, USA, 43–43. [ Cited on page 42. ]

[106] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Florence, Italy, 448–458. [ Cited on pages 77 and 115. ]

[107] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 691–701. https://doi.org/10.1145/2884781.2884807 [ Cited on pages 75, 76, 77, 80, 114, 115, 121, and 122. ]

[108] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 643–654. https://doi.org/10.1145/2884781.2884793 [ Cited on page 43. ]

[109] 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:1807.03837 [cs] [ Cited on pages 2, 9, 12, 41, and 87. ]

[110] 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 [ Cited on pages 1, 2, 10, 11, 12, 14, 15, 17, 23, 30, 35, 39, 41, 42, 56, 57, 59, 63, 72, 77, 119, and 121. ]

[111] 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 [ Cited on page 1. ]

[112] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. 2017. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '17)*. Association for Computing Machinery, New York, NY, USA, 28–35. https://doi.org/10.1145/3023956.3023966 [ Cited on pages 32, 33, and 120. ]

[113] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan. 2018), 1–24. https://doi.org/10.1145/3105906 [ Cited on pages 1, 2, 75, 90, and 114. ]

[114] Manish Motwani, Mauricio Soto, Yuriy Brun, Rene Just, and Claire Le Goues. 2020. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.2998785 [ Cited on pages 97 and 114. ]

[115] Saul B. Needleman and Christian D. Wunsch. 1970. A General Method Applicable to the

Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology* 48, 3 (March 1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4 [ Cited on pages 29 and 37. ]

[116] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. [ Cited on page 114. ]

[117] 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 [ Cited on pages 2, 12, 17, 41, 56, 63, 77, and 121. ]

[118] 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 [ Cited on pages 1 and 12. ]

[119] 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 [ Cited on pages 2 and 43. ]

[120] 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 [ Cited on page 52. ]

[121] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology* 5, 2 (April 1996), 99–118. https://doi.org/10.1145/227607.227610 [ Cited on pages 56 and 82. ]

[122] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An Experimental Evaluation of Selective Mutation. In *Proceedings of 1993 15th International Conference on Software Engineering*. IEEE, Baltimore, MD, USA, 100–107. https://doi.org/10.1109/ICSE.1993.346062 [ Cited on pages 56 and 82. ]

[123] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. 2014. HOMAJ: A Tool for Higher Order Mutation Testing in AspectJ and Java. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, Cleveland, OH, USA, 165–170. https://doi.org/10.1109/ICSTW.2014.19 [ Cited on page 71. ]

[124] 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 [ Cited on pages 53 and 71. ]

[125] 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 [ Cited on pages 1, 2, 49, 52, 61, 69, 71, and 72. ]

[126] Mike Papadakis and Nicos 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*. IEEE, Paris, France, 90–99. https://doi.org/10.1109/ICSTW.2010.50 [ Cited on page 50. ]

[127] 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 [ Cited on page 3. ]

[128] Spencer Pearson, Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, Buenos Aires, 609–620. https://doi.org/10.1109/ICSE.2017.62 [ Cited on page 52. ]

[129] 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 [ Cited on pages 49 and 56. ]

[130] Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René 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)*. IEEE, Vasteras, Sweden, 47–53. https://doi.org/10.1109/ICSTW.2018.00027 [ Cited on page 49. ]

[131] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-28901-1 [ Cited on page 43. ]

[132] 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 [ Cited on page 50. ]

[133] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254 [ Cited on page 80. ]

[134] 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 [ Cited on pages 2, 12, 14, 15, 39, 41, 59, 119, and 121. ]

[135] Manos Renieres and Steven P. Reiss. 2003. Fault Localization with Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, Montreal, Que., Canada, 30–39. https://doi.org/10.1109/ASE.2003.1240292 [ Cited on page 52. ]

[136] RTI. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Technical Report. National Institute of Standards and Technology. [ Cited on page 75. ]

[137] Stuart Russell and Peter Norvig. 2002. *Artificial Intelligence: A Modern Approach.* [ Cited on page 76. ]

[138] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 648–659. [ Cited on page 114. ]

[139] Seemanta Saha, Ripon k. Saha, and Mukul r. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 13–24. https://doi.org/10.1109/ICSE.2019.00020 [ Cited on pages 76, 83, 91, 92, 93, 95, 101, 114, 115, and 123. ]

[140] 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 [ Cited on pages 13, 17, 41, and 42. ]

[141] 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 [ Cited on pages 13, 17, 41, and 42. ]

[142] Julia Schroeter, Malte Lochau, and Tim Winkelmann. 2012. Multi-Perspectives on Feature Models. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12)*. Springer-Verlag, Berlin, Heidelberg, 252–268. https://doi.org/10.1007/978-3-642-33666-9_17 [ Cited on page 35. ]

[143] 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 [ Cited on pages 2, 11, 14, 22, 42, and 57. ]

[144] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. *ACM SIGPLAN Notices* 48, 6 (June 2013), 15–26. https://doi.org/10.1145/2499370.2462195 [ Cited on page 77. ]

[145] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 532–543. https://doi.org/10.1145/2786805.2786825 [ Cited on pages 90, 91, 97, and 114. ]

[146] 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 [ Cited on pages 12 and 41. ]

[147] 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 [ Cited on pages 11 and 42. ]

[148] 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 [ Cited on pages 13 and 42. ]

[149] 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. [ Cited on pages 11, 12, 41, and 42. ]

[150] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 727–738. https://doi.org/10.1145/2950290.2950295 [ Cited on pages 81 and 114. ]

[151] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches as Debugging Aids: A Human Study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 64–74. https://doi.org/10.1145/2635868.2635873 [ Cited on page 86. ]

[152] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (June 2014), 6:1–6:45. https://doi.org/10.1145/2580950 [ Cited on page 43. ]

[153] Susumu Tokumoto, Hiroaki Yoshida, Kazunori Sakamoto, and Shinichi Honiden. 2016. MuVM: Higher Order Mutation Analysis Virtual Machine for C. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Chicago, IL, USA, 320–329. https://doi.org/10.1109/ICST.2016.18 [ Cited on page 71. ]

[154] 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 [ Cited on pages 53 and 122. ]

[155] 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 [ Cited on pages 13 and 42. ]

[156] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation Analysis Using Mutant Schemata. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis - ISSTA '93*. ACM Press, Cambridge, Massachusetts, United States, 139–148. https://doi.org/10.1145/154183.154265 [ Cited on page 56. ]

[157] 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. [ Cited on pages 2, 14, 22, and 42. ]

[158] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! '14*. ACM Press, Portland, Oregon, USA, 213–226. https://doi.org/10.1145/2661136.2661143 [ Cited on pages 32, 33, and 120. ]

[159] 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 [ Cited on pages 13, 42, and 72. ]

[160] Lusheng Wang and Tao Jiang. 1994. On the Complexity of Multiple Sequence Alignment. *Journal of Computational Biology* 1, 4 (Jan. 1994), 337–348. https://doi.org/10.1089/cmb.1994.1.337 [ Cited on page 37. ]

[161] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Silicon Valley, CA, USA, 356–366. https://doi.org/10.1109/ASE.2013.6693094 [ Cited on pages 80 and 83. ]

[162] 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 [ Cited on pages 75, 76, 79, 80, 86, 87, 114, and 115. ]

[163] 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 [ Cited on pages 75, 76, 79, 80, 83, 93, 95, 114, and 115. ]

[164] Chu-Pan Wong, Jens Meinicke, Leo Chen, João P. Diniz, Christian Kästner, and Eduardo Figueiredo. 2020. Efficiently Finding Higher-Order Mutants. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 1165–1177. https://doi.org/10.1145/3368089.3409713 [ Cited on page 49. ]

[165] Chu-Pan Wong, Jens Meinicke, and Christian Kästner. 2018. Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 749–753. https://doi.org/10.1145/3236024.3264837 [ Cited on pages 7, 14, 42, and 56. ]

[166] 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 [ Cited on pages 2, 7, 17, 56, 57, 63, 77, and 94. ]

[167] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-Related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 660–670. [ Cited on pages 91, 93, and 95. ]

[168] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 789–799. https://doi.org/10.1145/3180155.3180182 [ Cited on pages 77, 87, 88, 108, and 115. ]

[169] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Buenos Aires, Argentina, 416–426. https://doi.org/10.1109/ICSE.2017.45 [ Cited on pages 114 and 115. ]

[170] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (Jan. 2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811 [ Cited on pages 77 and 93. ]

[171] 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 [ Cited on pages 13, 22, 41, 43, and 77. ]

[172] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (Oct. 2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648 [ Cited on page 93. ]

[173] 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 [ Cited on pages 11, 12, 41, and 42. ]

[174] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. 2007. Towards Locating Execution Omission Errors. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '07*. ACM Press, San Diego, California, USA, 415. https://doi.org/10.1145/1250734.1250782 [ Cited on page 42. ]

[175] 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, Piscataway, NJ, USA, 913–923. https://doi.org/10.1109/ICSE.2015.101 [ Cited on pages 1, 52, 53, 81, and 115. ]