On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems

Jens Meinicke,[1,2] Chu-Pan Wong,[2] Christian Kästner,[2] Thomas Thüm,[3] Gunter Saake[1]
[1]University of Magdeburg, Germany, [2]Carnegie Mellon University, USA, [3]TU Braunschweig, Germany

ABSTRACT

Quality assurance for highly-configurable systems is challenging due to the exponentially growing configuration space. Interactions among multiple options can lead to surprising behaviors, bugs, and security vulnerabilities. Analyzing all configurations systematically might be possible though if most options do not interact or interactions follow specific patterns that can be exploited by analysis tools. To better understand interactions in practice, we analyze program traces to characterize and identify where interactions occur on control flow and data. To this end, we developed a dynamic analysis for Java based on variability-aware execution and monitor executions of multiple small to medium-sized programs. We find that the essential configuration complexity of these programs is indeed much lower than the combinatorial explosion of the configuration space indicates. However, we also discover that the interaction characteristics that allow scalable and complete analyses are more nuanced than what is exploited by existing state-of-the-art quality assurance strategies.

Keywords: Feature Interaction, Configurable Software, Variability-Aware Execution

CCS Concepts: Software and its engineering → Feature interaction; Reusability;

1. INTRODUCTION

Highly-configurable systems challenge program analyses and quality assurance. Fault detection through testing becomes problematic as the additional dimension of configurability has to be considered: A test case that succeeds in many configurations may fail in others when configuration options interact [26, 28, 42]. Configuration faults are common in practice, but identifying interactions is difficult and challenging as the configuration space grows up to exponentially with the number of options [1, 26, 28, 42, 48, 67]. Unanticipated interactions can have consequences ranging from surprising behavior to security vulnerabilities and safety issues [27,52]. In practice, the configuration space is rarely analyzed systematically: ad-hoc testing of few or only one configuration is still common and testing of other configurations is often left to users [19,28,48]; at best, combinatorial interaction testing is used to check for interactions among pairs of options [53]. As a consequence, configuration faults are often only discovered by users [28,48].

Despite their exponential growth, there is hope for systematic or even complete analysis of configuration spaces. Investigations of bug reports have shown that most configuration faults are caused by interactions among only few options [1,17,21,26,42,47,53], and several analysis tools have successfully demonstrated exploitable redundancies among configurations to share commonalities while analyzing many configurations at once [5,11,13,34,37,39,43,51,61]. Our goal is to identify whether interactions occurring during program execution have characteristics that allow a complete analysis of the configuration space despite exponential surface complexity. In contrast to prior work that focused primarily on interaction faults (observable incorrect behavior) reported in practice (biased toward more popular configurations), we investigate all interactions in data *and* control flow to measure what we call *essential configuration complexity:* the configuration-related differences in an execution that actually need to be explored given an optimal execution strategy.

We designed a dynamic analysis tool, called VarexJ, that executes all configurations simultaneously and allows us to inspect differences in data and control flow at runtime. In particular, we use the tool to quantify the effort required for analyzing all interactions assuming (near-)optimal sharing. Analyzing executions of several medium-sized configurable Java applications, including Jetty and Checkstyle, we find that essential configuration complexity is indeed low enough to make a configuration-complete analysis feasible, but we also find that some common and important characteristics of interactions are not exploited by state-of-the-art analysis tools. We show that the driver for the complexity is not how many options interact, but how they interact on data. Our results help to understand the important characteristics for building efficient analyses for complete configuration spaces, and our dynamic analysis can help developers to understand how options interact within their system, possibly guiding their implementation toward reduced configuration complexity that is easier to understand and assure.

Overall, we contribute the following: (1) We implement a dynamic analysis for Java that tracks interactions on data and control flow during executing. (2) We develop three measures that characterize essential configuration complexity, measuring how options interact within an execution. (3) We design five benchmarks to study how state-of-the-art analysis approaches exploit interaction characteristics in exponential configuration spaces, exposing why certain approaches do

ASE'16, September 03-07, 2016, Singapore, Singapore

DOI: http://dx.doi.org/10.1145/2970276.2970322

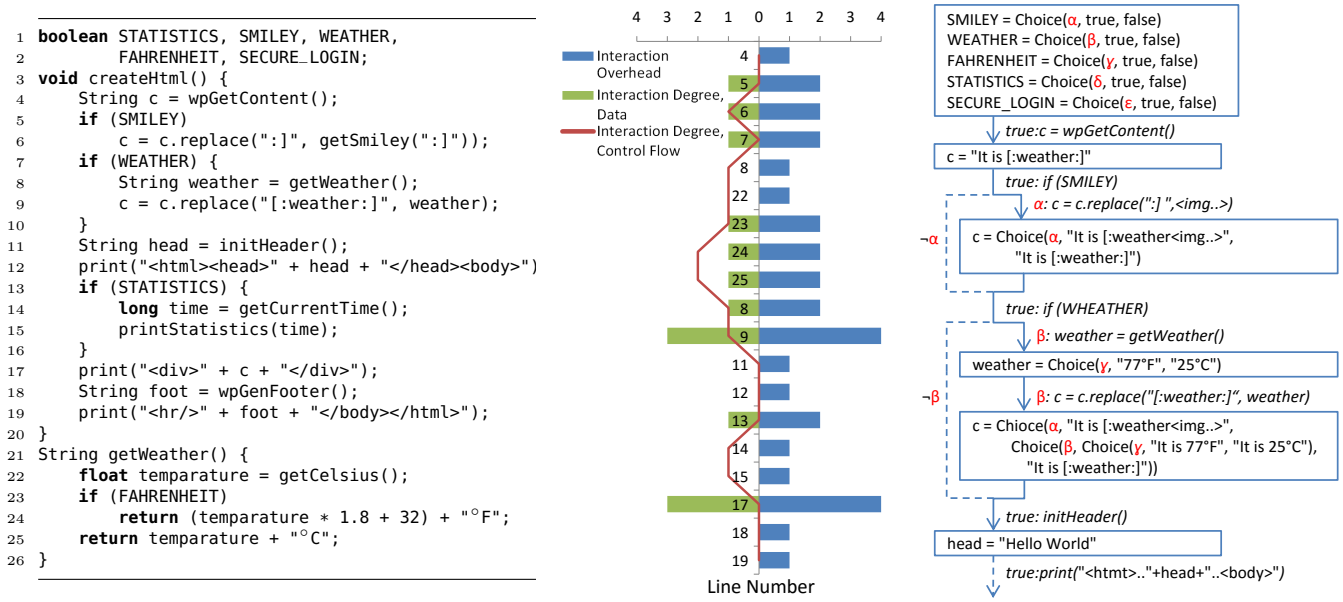

Figure 1: Feature interactions modeled after WorldPress. Example source (left), a measurement of feature interactions (middle), and the start of a corresponding variability-aware execution trace (right); descriptions at arrows display the executed statement with the corresponding context, boxes show changes on the state.

not scale for certain kinds of interactions. (4) We measure the configuration complexity for medium-sized systems, finding that essential configuration complexity is low enough to enable configuration-complete analyses. (5) We discuss common characteristics of interactions, providing more nuanced variants of current assumptions, which can, among others, encourage more efficient analyses for configuration spaces. The tool and further information can be found on our website http://meinicke.github.io/VarexJ/.

2. FEATURE INTERACTIONS

A feature interaction describes the situation in which features modify or influence another feature in describing or generating the system's overall behavior [15, 68], typically observable when the combined behavior of two features differs from the individual behaviors of both features [18, 52]. For example, one feature can interfere with or overwrite the effects of another feature. When both features are developed independently, it can be difficult to predict such interactions. The concept of feature interactions has gained attention in 90s when studied in the context of telecommunication systems [18]. In general, a feature interaction is a failure of compositionality, in the sense that the developers of features did not anticipate the interaction when combining them.

Feature interactions are a common problem in software systems with configuration options and systems that are composed of different modules (e.g., plugins, components); they can lead to often behavior, bugs, and security vulnerabilities [1, 26, 42, 51–53]. In the remainder of the paper, we refer to features, optional modules, and options in a software system uniformly as *configuration options* and refer to a specific combination of those as a *configuration*. Interactions may occur only in specific configurations and are thus difficult to detect: A product working fine in one configuration may exhibit unexpected behavior when changing an unrelated option or installing a new plugin. The exponentially growing configuration space challenges quality assurance. At the same time, isolating options to prevent interactions is often not possible, because some options are designed to interact (e.g., intentionally exchanging specific kinds of data among plugins or apps). The challenge is to allow intended interactions, but prevent or detect accidental ones.

As example of interactions among multiple options consider the code excerpt in Figure 1, modeled after the WordPress blogging software that is extensible by thousands of independently developed plugins. Options *weather* and *fahrenheit* interact purposefully to show the weather information in desired format. However, options *smiley* and *weather* can interact on the blog post's content in unintended ways, such that the *smiley* code breaks *weather*'s expansion of a '`[:weather:]`' tag, rewriting it into '`[:weather☺`'.

There are different definitions of what constitutes an interaction, depending on what characteristics can be observed. *Interaction faults* tend to refer to issues in which we can observe a fault (e.g., a crash) in an execution if and only if options are combined in specific ways [1, 26, 53]. For the purpose of our discussion, we understand as an interaction *any trace or state difference* in the program that depends on two or more options, even if it does not result in a crash or even observable behavior. For instance, in our WordPress example, whether and how statistics are printed depends only on option *statistics* (not an interaction), but the value of `c` depends on *smiley*, *weather*, and *fahrenheit* (data interaction) and whether Line 24 is executed depends on both *weather* and *fahrenheit* (control-flow interaction). We consider interactions at the low level of data and control flow, because they are measurable even without a specification of intended behavior. With the *degree of an interaction*, we refer to the number of options involved in the interaction; typically, interactions of higher degree occur in fewer configurations and are harder to detect.

Our goal is to understand which interactions exist at all in software systems to inform the demands on quality assurance and specification techniques that can be used to detect interactions and to distinguish expected behavior from faults.

Surface vs. Essential Configuration Complexity. On the surface, analyzing the entire configuration space seems unrealistic because it grows exponentially with the number of configuration options. A brute-force approach executing each configuration separately is conceptually *configuration complete*. That is, it explores all variations, but it is only feasible in practice for the smallest configuration spaces.

Fortunately, the amount of variability that actually induces differences and interactions in the execution may be small enough to handle with a suitable analysis strategy. We distinguish between *surface configuration complexity* (exponential explosion with the number of options in the system) and *essential configuration complexity* describing the actual differences caused by options and their interactions.[1] For instance, in our example only three of five options affect the blog post content and no option ever affects the header.

We aim to capture essential configuration complexity with a configuration-complete dynamic analysis that allows us to analyze which options really interact and how. In addition, we look for common characteristics of interactions that can be exploited by quality assurance strategies and developers.

Quality Assurance for Large Configuration Spaces. Several quality-assurance strategies have been developed for detecting interactions in large configuration spaces [19, 61]. We focus on execution-level approaches that can assure a specific execution with given inputs over many configurations—for example, checking whether a test case for an option's behavior, such as replacing smilies by images, passes independently of other options.

State-of-the-art quality assurance strategies for large configuration spaces typically use one of two strategies to scale:

- *Sampling-based strategies* execute the program only in selected configurations and are thus *not configuration complete*. That is, they may miss interactions. Systematic sampling strategies, such as combinatorial interaction testing, can explore all interactions up to a given degree, but may miss interactions with a higher degree [53]. Their underlying assumption is that the essential complexity is much lower than the surface complexity in that most configuration options are orthogonal and most interactions are of a low degree [1, 17, 21, 26, 42, 47, 53].
- *Sharing-based strategies* exploit lower essential complexity more directly through the observation that many executions in different configurations are similar or even identical. They attempt to reduce redundancies by sharing part of the execution. In the simplest case, identifying that an option is not used (e.g., option *secure_login* in our example), a configuration with and without that option will have identical traces [37, 39]. More sophisticated approaches often build on top of heavy-weight infrastructures such as model checkers and may share a prefix of the execution and fork it only once a computation depends on variability, possibly even joining again to share subsequent executions [5, 51, 55, 57]. Sharing-based strategies tend to reduce unnecessary surface complexity, often while remaining *configuration complete*.

The effectiveness of both sampling-based and sharing-based strategies depend on the essential configuration complexity as well as the effectiveness of the sampling and sharing mechanisms. Different approaches exploit different characteristics of interactions, but there is little knowledge of actual characteristics. The prevalence of low-degree interactions is supported by observations of reported interaction faults [1, 17, 21, 26, 42, 47, 53], but such results are potentially biased, because faults in more popular configurations are more likely to be reported. In addition, incomplete sampling-based strategies are more readily available in practice and may primarily find faults supporting their assumptions of low interaction degrees. Redundancy during execution has been successfully exploited by several approaches [37–39,63], but scalability has been limited and it is unclear whether that is due to the essential configuration complexity or insufficient exploitation of redundancies.

3. MEASURING CONFIG. COMPLEXITY

To assess configuration complexity and find interaction characteristics, we need a way to identify interactions in a software system's execution. Using a brute-force approach, we could attempt to record traces (and state) of the system in all configurations [70] and subsequently compare traces [31, 65] to assign differences to options or interactions, but scalability concerns would restrict us to systems with only few options. Similarly, a sampling strategy is not suitable, because we might not find rare high-degree interactions that could be part of the essential configuration complexity. Instead, we designed a dynamic analysis that tracks all interactions during an execution using a sharing-based strategy.

Our dynamic analysis coordinates the execution of all configurations, which allows us to observe which statements and values actually differ among configurations during the execution. Our analysis aggressively exploits sharing, but has a high constant overhead and high engineering costs (modifying an interpreter to track state and control flow of multiple configurations at once; giving up features, such as just-in-time compilation). Ideally, our results can inform the design of future tools that can exploit the important characteristics of interactions in a simpler and faster way.

Our dynamic analysis is built on the idea of variability-aware execution [6, 36, 51]. Specifically, we have implemented VarexJ [49], a variability-aware interpreter for Java Bytecode and instrumented it to record interactions. In the following, we discuss how variability-aware execution works and how we use it to assess essential configuration complexity.

3.1 Variability-Aware Execution

A variability-aware interpreter aims to maximally share redundant executions using *conditional values* and *variability contexts*, at the cost of additional overhead for each computation [36, 49, 51]. A conditional value is a multi-value that may have different concrete values in different partial configuration spaces [25, 64]. A variability context is a formula that describes a partial configuration space. Let us illustrate conditional values by example:

- $Choice(\alpha, x, y)$ is the representation of two alternative concrete values: x for all configurations in variability context α, and y for all others.
- $Choice(\alpha, x, Choice(\beta, x, y)) = Choice(\alpha \vee \beta, x, y)$ illus-

[1] Our notion of surface and essential configuration complexity is inspired by Brook's discussion of accidental and essential complexity in software engineering [14], in which essential complexity describes the unavoidable complexity of the problem overshadowed by accidental complexity from suboptimal languages, tools, and processes.

trates how choices can be nested to represent more than two alternative values, and how they can be compressed to store only distinct values.

To compute with conditional values, we have to apply operations to all alternative values. That is, we need to consider all valid combinations of values (i.e., the cross product in the worst case). For example, we compute the sum of two conditional values and compress the result as follows:

$$Choice(\alpha, 0, 1) + Choice(\beta, 0, 1)$$
$$= Choice(\alpha, Choice(\beta, 0, 1), Choice(\beta, 1, 2))$$
$$= Choice(\alpha \wedge \beta, 0, Choice(\alpha \vee \beta, 1, 2))$$

Some computations may be performed only within selected configurations. To this end, a variability-aware interpreter keeps track of the *variability context* in which each instruction is executed (similar to path conditions in symbolic execution).

The interpreter keeps track of all data using conditional values, which enables a fine-grained representation of shared data. If, for example, the value of a field differs among configurations, the values are stored as a choice in the field, but other fields of the same object are shared for the entire configuration space. Instead of splitting the entire heap, variations are stored locally. When computing with data, we only have to compute with distinct values of all inputs, of which there are typically much fewer than configurations in the configuration space. Furthermore, the compact representation using variability contexts in choices provides us with a way to track where options interact.

Note how options occur only in variability contexts of choices, but all values are concrete. In contrast to symbolic execution, symbolic configuration decisions do not intermix with concrete values. Hence, all computations are performed with concrete values. This separation of concrete and symbolic values enables computations without the undecidability issues from abstractions in symbolic execution, therefore we rely on variability-aware execution in our study.

We illustrate how a variability-aware interpreter executes all program configurations of our example with the partial trace in Figure 1 (right). The five configuration options are initialized to both `true` and `false` with a condition. Subsequently, the statement in Line 4 is executed once for all configurations (context *true*). The *if* statement splits the execution, such that Line 6 is executed in a restricted context (α for *smiley*), creating a choice in the heap for variable `c`. After the if block, the statements can be shared again for all configurations. In Line 8, the method `getWeather` is called, returning a choice depending on the option *fahrenheit*. Variability-aware execution only needs to invoke the method once, returning a choice as result. By applying the calculated weather to `c`, the string replacement needs to be performed four times, creating a choice that depends on three options with four distinct values. Finally, the code from Line 11 can be shared again for all configurations, until code depends on conditional data again, as in Line 14 and 17, respectively.

A variability-aware interpreter maximizes sharing of redundant calculations in two ways: First, variability-aware execution moves on from one instruction to the next instruction sequentially only when every possible concrete value has been computed for the corresponding configuration space. In other words, variability-aware execution achieves instruction-level sharing among control flows of all possible configurations. Second, the difference between program states is represented compactly using choices, such that small differences in local variables or heap objects can be represented without splitting the entire program state. In this way, a variability-aware interpreter achieves fine-grained sharing among all executions.

Implementation. We implemented our variability-aware interpreter VarexJ on top of JavaPathfinder's [30] interpreter for Java Bytecode. To implement variability-aware execution, we modified all bytecode instructions to handle conditional data, we extended all shared data structures (e.g., the heap, the method frame) to store choices, and we implemented a specialized scheduling mechanism.

Those changes and the fact that VarexJ itself is written in Java creates a high runtime overhead for each instruction (a constant slowdown compared to a JVM of a factor 50–250 in our experience). As we build on top of JavaPathfinder, we inherit the same limitations, such as incomplete support for native methods and limited support for concurrency. This overhead and these limitations might forbid using VarexJ for practical testing, but it is acceptable for our explorations of configuration complexity. The advantage of extending an interpreter is that we can monitor each Java Bytecode instruction to observe interactions during runtime. To ensure the correctness of the implementation, we compared the executed instructions to the execution of all single configurations for several of our subject systems (cf. Section 5).

More detailed descriptions of variability-aware execution and our implementation can be found in the first author's master's thesis [49] and on our website.

3.2 Measuring with VarexJ

Sharing executions and compactly representing data differences, our dynamic analysis can directly collect data about interactions. Our execution overapproximates essential interaction complexity where sharing is suboptimal. Technically, we instrumented the execution of each Java bytecode instruction to collect data on interactions to measure three metrics: the control-flow interaction degree, the data interaction degree, and the interaction overhead. We exemplify the measurements for our running example in Figure 1 (center).

With *control-flow interaction degree*, we measure configuration complexity on the control flow by assessing how many options need to be selected or deselected to execute the instruction at this point of the trace. The degree increases at control flow decisions that depend on a configuration option; in our example, the instruction in Line 24 is executed with context *weather*$\wedge$*fahrenheit*, thus this instruction's control-flow interaction degree is two. As our analysis already tracks the variability context during execution, we merely need to log the number of options in the context for each executed instruction. In our plots, we visualize the control-flow interaction degree as a red line along the trace. A high value indicates part of an execution that is only triggered in few configurations.

With *data interaction degree*, we measure configuration complexity on data by assessing on how many options the resulting value of an instruction depends. Considering variability, an instruction may need to be computed with alternative values and the result of the instruction may depend on one or multiple options, of which we report the number of distinct options affecting the value. For example, the expression computing `c` in Line 9 results in four alternative values depending on three different options, resulting in a data interaction degree of three. We measure the degree by inspecting the result of every instruction during execution and plot it as a green bar along the trace. A high value indicates that some different results from a computation might be observable in few specific configurations only.

Finally, with *interaction overhead*, we measure the effort

required to execute an instruction considering data variability in the instruction's inputs. If all inputs of an instruction have the same value in all configurations, we need to execute the instruction only once (baseline overhead 1). If one input has n alternative values in different configurations, we need to execute the instruction n times (overhead n). For instructions with multiple inputs (e.g., addition or method invocation), we need to consider all combinations of alternatives of all inputs (worst case overhead $n \times m$ for an instruction with two inputs with n and m alternatives respectively). In contrast to interaction degree measures, interaction overhead assesses the essential computational effort from alternative values, not how many options are involved. For example, in Line 9 the two values of `weather` are combined with the two values of `c` (overhead 4). We compute the interaction overhead by inspecting the variability in all inputs of each instruction and plot it as blue bars along the trace. The interaction overhead is useful to compare essential complexity to the effort for executing a single configuration; comparing the aggregated overhead of all instructions with those of a single execution allows us to assess how many additional instructions have been executed and how many instructions need to be repeated due to variability.

All three measures assess different aspects of configuration complexity. The interaction degree measures characterize interactions in control flow and data, whereas interaction overhead approximates the effort required for a configuration-complete analysis considering maximal sharing. The trace for our example in Figure 1 illustrates how the measures peak every time data from interactions is created or accessed.

4. INTERACTION BENCHMARKS

After introducing how we measure configuration complexity technically, we illustrate how certain kinds of interactions affect essential configuration complexity with a series of benchmarks. The benchmarks provide a sanity check for our measures of configuration complexity before we collect and interpret the measures on real-world systems. Additionally, they allow us to study how well existing sharing-based analysis tools exploit redundancies and which interaction characteristics they exploit. This enables us later to extrapolate which analysis strategies can cope with characteristics found in real-world software systems.

We designed five benchmarks shown in Table 1 that each exhibit different interaction characteristics in a short execution. In the second column, we plot the measured configuration complexity. Additionally, we compare execution time, executed instructions, and memory consumption of five state-of-the-art analysis tools: SPLat [39], JPF-Core [30], JPF-BDD [63], JPF-SE [2], and VarexJ. We do not evaluate sampling-based strategies, as our benchmarks are specifically designed to produce high-degree interactions. Specifically, we address the following research question: **RQ 1: What are the effects of different kinds of interactions on the scalability and performance of state-of-the-art execution mechanisms?**

4.1 Experimental Setup

Evaluated Analysis Tools. We compare five state-of-the-art analysis tools that have been designed to efficiently execute a program over configuration spaces by fighting surface complexity through different kinds of sharing. Some of these tools have been designed originally for different purposes, such as model checking safety properties [2], but they have been suggested also for analyzing interactions or testing highly-configurable systems. We selected tools that represent different analysis and sharing strategies: identifying unnecessary options, software model checking, and symbolic execution. In addition, we use the uninstrumented version of our variability-aware interpreter VarexJ as a representative for variability-aware execution. The tools are comparable in the sense that they all target Java and are mostly based on the same infrastructure, namely JavaPathfinder [30].

JPF-Core (JavaPathfinder) is a model checker for Java Bytecode that handles bytecode instructions as transitions between states [30]. JPF-Core can be used to split execution paths for boolean options and explore all possible paths. If all values of fields and variables are equivalent, JPF-Core can join separated paths and share subsequent executions.

JPF-BDD extends JPF-Core by separating tracking of boolean options [63]. By taking options out of the state, states can be merged if they differ only by options, increasing the chance for joining, and thus sharing.

JPF-SE is a symbolic extension of JavaPathfinder [2], designed for test generation. If a variable is assigned with a symbolic value, the search tree splits, but due to challenges in matching symbolic states, states are never merged.

Finally, SPLat instruments a program to dynamically detect which configuration options are used in an execution [39]. It reexecutes the program until all combinations of used options are explored. Although SPLat does not share any actual executions, it can narrow down the configuration space if only a subset of configurations have an effect on the execution trace (e.g., for unit tests). As the tool is not publicly available, we reimplemented it for Java.

Benchmarks and Metrics. We design five small benchmark programs characterizing favorable and critical cases for interactions among configuration options. We show all benchmarks in Table 1 and explain them and their rationale together with the results. All benchmarks are reduced to distill the interaction effect in a very concise setting. Each benchmark can be scaled in the number of involved configuration options, such that we can observe scalability with regard to the exponentially growing surface configuration complexity. We plot the *complexity measures* for an execution with 10 options to illustrate the general trend.

For each tool, we report the *performance measures* time, instructions, and memory consumptions for executing the benchmark with different numbers of options (0 to 100). We measured them all using internal metrics of JavaPathfinder and built a separate harness for SPLat. As we face an exponential problem, we terminate executions that exceed two minutes. To reduce measurement bias, we report the average of three runs.

4.2 Sharing Potential

For each benchmark, we discuss the interaction characteristic it simulates, reasons for the configuration complexity, and the performance measures indicating which tools scale.

B1: Explosion. We start with the worst case of interactions in which all options interact on the same value and yield a different result in every configuration. In such case, every exhaustive technique needs to track an exponential number of alternative values. As visible from the complexity measures, early and some later instructions (e.g., if statements) in the benchmark are affected by fewer configurations and can be shared. However, no tool can be expected to scale as they all face es-

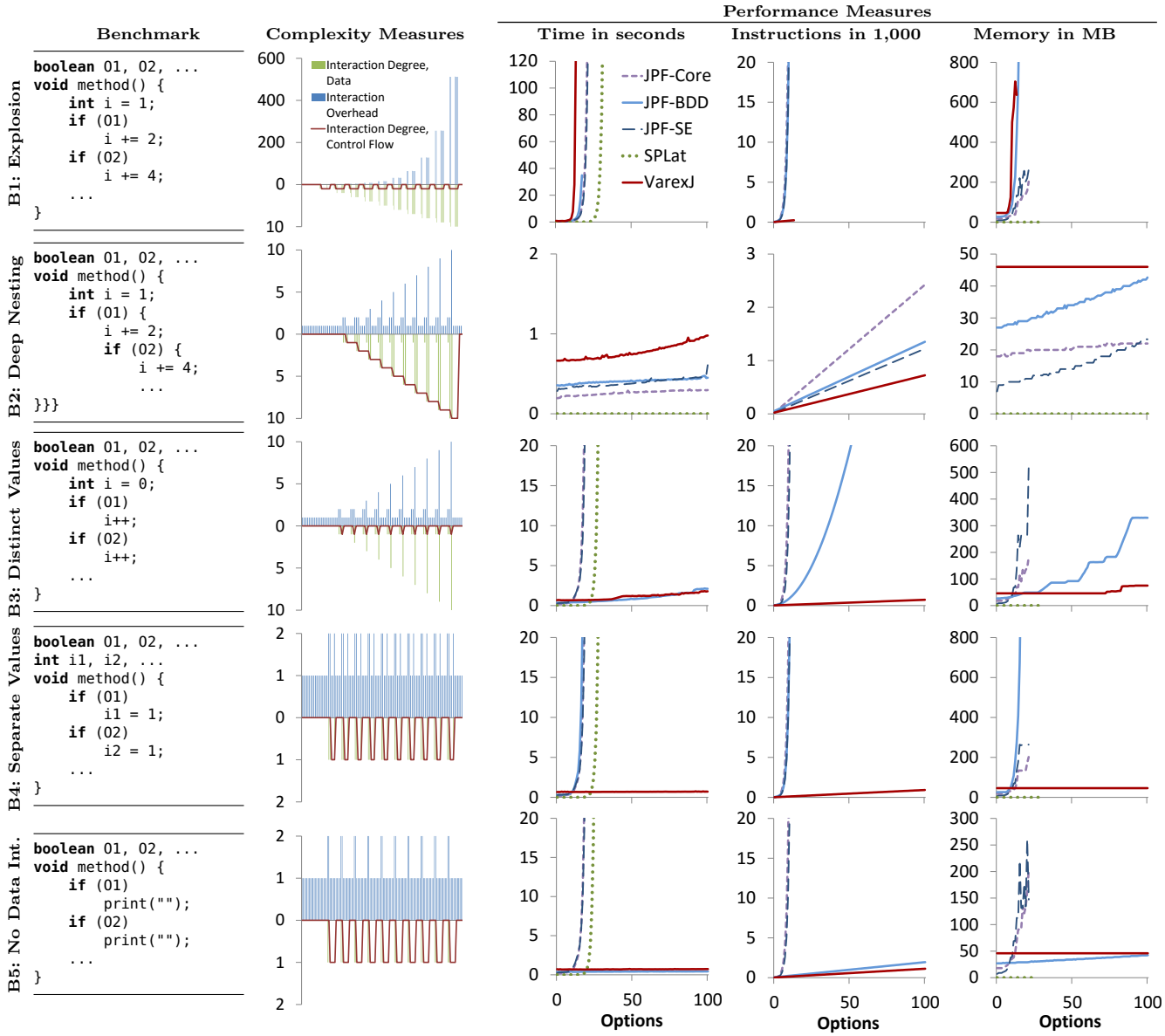

Table 1: Benchmarks to simulate different kinds of interactions (left). The diagrams in the second column illustrate interactions for each program measured using variability-aware execution. The three diagrams on the right show the performance results for time, executed instructions and memory consumptions for five analysis tools.

sential configuration complexity growing exponentially with the number of options as visible in all performance measures. If these kinds of interactions are common in practice, there would be little hope for configuration-complete analyses.

B2: Deep Nesting. Next, we explore the effect of dependencies among options, leading to a lower essential configuration complexity with a linear number of distinct execution traces. Whereas B1 had independent decisions for each option, resulting in 2^n execution traces, B2 models nested decisions, resulting in $n+1$ execution traces for n options. The complexity measure shows the linear increase in overhead as additions are performed on values with increasingly many alternative values. Also the interaction degree measures grow linear as more and more options need to be selected. Again, we can see that several instructions that do not manipulate variable i could be shared. Our performance measures show that this kind of interaction is well supported by all approaches. As all tools only split lazily where necessary, there are nearly linear increases with more options in all performance measures. Simpler tools outperform tools with higher constant overhead, as exploiting additional sharing has only marginal effects.

B3: Distinct Values. Sharing becomes feasible if interactions on a variable produce a small number of distinct values. In benchmark B3, each option increases a value by 1, resulting in $n+1$ distinct values for n options. Therefore, essential configuration complexity grows linearly with the number of configurations. Our performance measurements indicate that JPF-Core and JPF-SE require exponential effort as they need to split the execution on every *if* statement but cannot join them again; JPF-SE never joins and JPF-Core cannot join as the values representing the options have different values in different states. Without data sharing, also

SPLat requires exponential effort because all options have an independent effect on the execution trace. JPF-BDD and VarexJ both track the $n+1$ distinct values separately from the variations in configuration options, which enables them to perform closer to the linear growing essential configuration complexity. VarexJ executes fewer instructions and requires less memory than JPF-BDD by exploiting additional sharing, which however has no benefits for the execution time due to the additional overhead.

B4: Interactions on Separate Values. If options affect disjoint parts of the state, essential configuration complexity can be very low. Benchmark B4 exhibits an interaction in which each option affects a different variable, without any data interaction. Despite an exponential number of execution traces and distinct states, each variable has only two alternative values (0 and 1) and, as such, the essential configuration complexity is low. As the performance measures show, JPF-Core, JPF-SE, JPF-BDD, and SPLat all require exponential effort, as they do not exploit sharing for this interaction characteristic. All approaches split on each if-statement and none can join the states again. Even JPF-BDD cannot join, as non-option values differ across configurations. Only VarexJ approaches the low essential complexity.

B5: No Interactions on Data. Finally, we eliminate all data interactions, such that only control-flow interactions remain (i.e., an exponential number of different execution traces, all with the same states). Essential configuration complexity is low as in B4. JPF-BDD and VarexJ both execute each instruction on a single state without interaction overhead, as all variability of options is handled separately. In contrast, SPLat still needs to explore all execution traces and JPF-Core and JPF-SE track different configuration values as part of their split state, resulting in exponential behavior.

Lessons Learned. Even when essential configuration complexity is low, missing to exploit suitable forms of sharing for certain characteristics of interactions can result in exponential execution efforts. A program with negligible essential complexity (e.g., without any data interaction, as in B4) can cause exponential behavior in state-of-the-art approaches. Finding such kind of interaction characteristics in real-world programs would be a great opportunity for quality assurance, as it indicates a high potential for configuration-complete analysis with sharing-based approaches.

5. REAL-WORLD INTERACTIONS

To assess essential configuration complexity of executions in real-world software, we applied variability-aware execution to eight configurable systems shown in Table 2. We selected four configurable medium-sized systems from different domains, the http server Jetty 7, the in-memory database Prevayler, the static analysis tool Checkstyle, and the academic evaluation framework for database index structures QuEval [56]. In addition, we included systems previously used as benchmarks in research on configurable systems: The systems MinePump [41], E-Mail [29], and Elevator [54] are small academic Java programs that were designed with many interacting options; GPL [45] is a small-scale configurable graph library often used for evaluations in the product-line community. All these systems are executable with VarexJ.

To investigate interactions in configurable systems, we pose the following research question: **RQ 2: What is the essential configuration complexity of real-world software?** Particularly, we are interested in whether our measures for configuration complexity confirm current assumptions based on error reports and program outputs [1, 26, 42] or whether they provide additional insights.

System	LOC	Opt.	Conf.	Inst.VA	$\sum$IO	μInst.	Cov.
Jetty 7	145,421	7	128	12M	12M	12M	16%
Checkstyle	14,950	141	$>2^{135}$	407M	421M	198M	37%
Prevayler	8,975	8	256	28M	29M	15M	7%
QuEval	3,109	20	680	81M	94M	1M	45%
Elevator	730	6	20	89k	100k	29k	81%
GPL	662	15	146	17M	17M	9M	86%
E-Mail	644	9	40	48k	55k	16k	96%
MinePump	296	6	64	14k	16k	14k	84%

Table 2: Subject systems analyzed for configuration complexity and their sizes in lines of code, number of options and configurations; number of instructions executed with VarexJ, the aggregated interaction overhead ($\sum$IO), the average number of instructions for single configurations (μInst.), and lower bound for line coverage reached with the sample method.

Experimental Setup. We execute all subject systems over all configurations with VarexJ. For each system, we measure configuration complexity for a fixed standard input: a sample input distributed with QuEval, a source file with 474 lines for Checkstyle, and a sample application provided with Prevayler. For Jetty, we deploy a web application that is capable of serving static content as well as running simple servlets. As the traces often contain several million instructions, we aggregate (max) subsequent instructions in our plots. We share the evaluation setup together with our implementation.

Interactions in Real-World Software. We show five representative traces in Figure 2; the remaining traces can be found on our website. In all systems, we can observe a small average interaction overhead throughout most of the trace and usually small interaction degrees (i.e., most instructions can be shared in large configuration spaces). The traces also show that options do not interact increasingly across the entire executions. Some individual results are noteworthy:

First, the Elevator system was specifically designed to exhibit many interactions [54]. Its trace shows that several interactions on data cause an interaction overhead of up to 12. However, most instructions in the trace have an overhead of at most two. Many instructions are executed in restricted contexts though, requiring up to five options.

Second, GPL is a common system for evaluations in the product-line community, including prior studies of sharing and verification [5, 38]. The system has only some minor interactions with an interaction overhead of mostly two and a interaction degrees of mostly one option. Options do not interact at all for most parts of the trace. However, at the end of the execution up to eight options interact on the same data.

Third, we observed the strongest data interactions in QuEval. QuEval implements several database index structures which can be customized with several options, significantly changing the behavior of the entire system. The trace shows that there are long sequences with similar overhead in the execution. This is caused by separate processing of each index structure. Some values interact strongly causing an overhead of 100 (among 680 configurations). However, the trace still shows that high interaction degrees are rare, and many instructions can be shared after and between them. In QuEval, there are multiple interactions that cause high interaction degrees on data and control flow. Especially, in

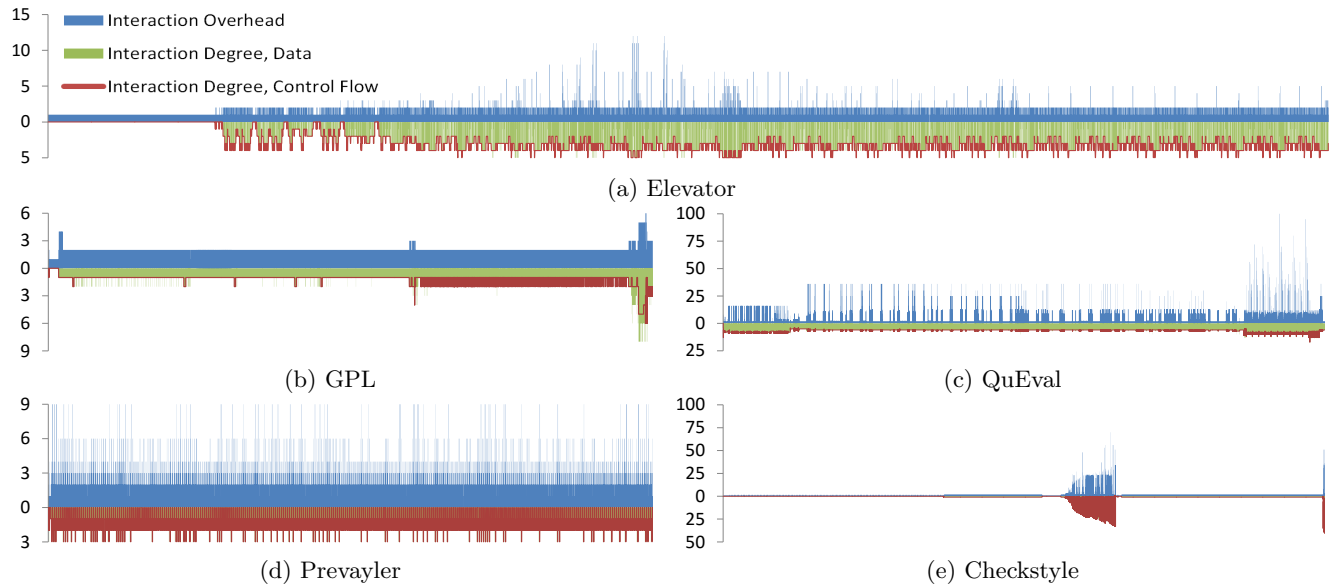

Figure 2: Traces and interaction overhead of variability-aware execution for larger software. Each bar represents the highest value per 1,000 instructions (per 10 for Elevator).

the last part of the execution, data interactions similar to benchmark B1 can be observed for a subset of the options.

Fourth, Checkstyle is a good example for a trace with particularly few interactions. The system implements many optional and independent checks that are not supposed to interact. However, the trace shows that there are still high degree interactions in Checkstyle, mostly caused by optional caching, resulting in a similar behavior as in the benchmark B3 in a subset of the trace. Also in Jetty (trace not shown), we similarly observe that most options have only minimal influences on the trace; we found no interactions on data at all.

Throughout all systems, we observe essential configuration complexity that is far lower than surface complexity may indicate. The amount of essential configuration complexity differs by system though from almost negligible (Checkstyle, Jetty, GPL), to medium (Elevator, Prevayler), to significant (QuEval). Comparing the aggregated interaction overhead with the average number of instructions executed without variability shown in Table 2 (cf. Sec. 3.2), we can see that a system executing close to essential configuration complexity would usually only have to execute $1 - 4$ times more instructions than an average execution of a single configuration. Only QuEval had a significant interaction overhead compared to an average execution of individual configurations, but that can be explained largely by executions for alternative options. In general, the overhead is much lower than the overhead of factor 20 to 2^{135} a brute-force approach would require and could potentially even beat some sampling strategies that reexecute each sampled configuration.

In its current form, due to the high overhead per instruction, VarexJ cannot achieve this speedup compared to a standard JVM. [2] However, our results indicate that essential complexity is low and there is hope for the community to develop efficient configuration-complete analysis techniques.

Threats to Validity. Concrete results from our measurements should be generalized only carefully; our focus is on establishing metrics for configuration complexity, not on proving characteristics of programs in general practice. External validity is limited by the number and size of our subject systems. As described, we selected the small programs representing critical and paradigmatic cases, whereas we used convenience sampling for the medium-sized systems, primarily due to current technical limitations of our interpreter and the high engineering effort to execute further and larger systems. Our subject systems are diverse, but their characteristics may not generalize for other systems.

As described, we executed each system with only one input. Thus, we potentially miss interactions that occur only with other inputs. Nonetheless, we execute each program's main method with a representative input, which in each system covers all configuration options and a large amount of its code as the measured line coverage in Table 2 indicates.

To interpret our results, it is important to remember, as discussed in Sec. 2, that we define interactions as any differences during the execution triggered by options, not just externally visible differences or defects. This decision is deliberate to study interactions and execution methods in general, independent of defects they may cause.

6. DISCUSSION: CHARACTERISTICS OF INTERACTIONS

In Section 4, we have shown that despite exponential surface complexity many kinds of interactions actually have low essential complexity, which can be exploited by suitable sharing-based analyses. In Section 5, we have subsequently shown that also real-world systems typically have a much lower essential complexity than it may appear on the surface. However, we have also seen that interactions in real-world systems have characteristics that are more nuanced than ex-

[2] When compared to executing a single configuration with VarexJ's own interpreter, we observe performance overheads between 1.0x (Jetty) and 9.7x (GPL) for most systems and 190x for QuEval, in line with the measured interaction overhead. Detailed performance measurements are outside the scope of this paper, but can be found on our website.

pected by existing approaches. Therefore, we conclude with a discussion of observed characteristics that may inform the design of future analysis approaches and may also be informative for developers concerned about interactions in their code.

We identify three main characteristics that are exploited (though not always explicitly) by existing analyses: irrelevant variability, orthogonal variability, and local variability.

Irrelevant variability. Some options may not have any effect on an execution at all. Even when a program has a large configuration space, some executions, such as test cases, may not even read certain configuration options. If no configuration of an execution ever reads a configuration option, we call such execution **unaffected** by the option (e.g., option *secure login* in our example of Figure 1). In addition, some options may never be read unless another option is (de)activated, in which the first option **depends** on the second (e.g., *fahrenheit* depends on *weather* in our example). In both cases, the number of distinct executions is smaller than the exponential surface complexity indicates.

All sharing-based approaches exploit irrelevant variability, as shown with benchmark B2. Although, irrelevant variability was attributed with significant speedups for test cases in prior work [37,39], none of our real-world executions benefited from *unaffected* variability without rewriting the system to initialize options lazily (all options were always read and initialized). *Dependencies* reduced the search space, but never close to essential configuration complexity.

Orthogonal Variability. Many options may not interact with each other. Although potentially every option could interact with every other option, resulting in exponential surface complexity, a common assumption is that most options do not interact. We say two options are orthogonal if combining both options does not yield any new behavior that could not be explained by either option alone. Some options may be **strictly orthogonal** and not interact with any other option (e.g., option *statistics* is strictly orthogonal to all other options in our example), but it is more common to assume **low interaction degrees** where each change can be explained by the interaction of at most two or three options (e.g., options *smiley*, *weather*, and *fahrenheit* all interact on the blog post, but not with any other options).

The effectiveness of sampling strategies typically hinges on low interaction degrees (see Sec. 2), whereas most existing sharing-based approaches are rather inefficient in exploiting orthogonality, especially when options affect data, as apparent from benchmarks B4 and B5. Our real-world executions confirm that many options are orthogonal, but also show that one should not rely on low interaction degrees alone: We found high interaction degrees (e.g., 40 in Checkstyle) in most systems, but also found that those involve some options while others remain mostly orthogonal. We argue that **rare high interaction degrees** is a more accurate characterization of interactions in real-world systems, encouraging research into configuration-complete analyses.

Local variability. An option may affect control flow and data during an execution, but its effects might not spread across the entire execution trace, resulting in much lower essential configuration complexity than surface complexity. With locality, we might need to invest more effort to execute part of the trace repeatedly for different configurations, but we can share effort in other parts. In our example, option *statistics* produces additional output, but does not affect earlier or subsequent instructions, neither through control flow nor through data in stack or heap.

	Unaffected	Depending	Strictly Orthogonal	Low Interaction D.	High Interaction D.	Prefix Sharing	Strictly Local	Scattered Local	Eng. & Runtime Ov.
Combinatorial testing			●	●					very low
SPLat	●	●							low
JPF	●	●	◍	◍	◍	●	◍		high
JPF-SE	●	●	◍	◍	◍	●			very high
JPF-BDD	●	●	◍	◍	◍	●	●	◍	high
VarexJ	●		●	●	●	●	●	●	very high

Table 3: Interaction characteristics exploited by different analysis approaches. ●: exploited, ◍: partially exploited (e.g., JPF-BDD exploits scattered local only for boolean values.)

Many sharing-based approaches exploit locality by sharing executions before the option's effect, and possibly also after (see Sec. 4). Existing sharing-based approaches differ in what forms of locality can be exploited though. Many approaches can share a common prefix of the execution trace (**prefix sharing**) and split late on the first instruction depending on an option. Some approaches can join after local instructions, if those instructions do not affect the state as option *statistics* in our example and in benchmark B5 (**strictly local**). Interactions that affect some state, that is, however, not read again subsequently (see benchmark B4) are rarely supported.

A much more common pattern in the observed real-world executions is what we call **scattered local**: Options affect the trace locally and cause some changes to the program's state, but many subsequent instructions can be shared before that changed state is accessed again. In our example, multiple options affect the value of `c`, but subsequent instructions can be shared until `c` is read again at the end of the method. This is an effect, which we observed as gaps between peaks in the measures of our benchmarks and the real world executions. In all cases, we see strong evidence of locality in that essential configuration complexity always returns to lower values after peaks.

Outlook. We observed that essential configuration complexity is often low and exploiting irrelevant variability, orthogonal variability, and local variability is a promising avenue to scale analysis approaches. However, we also found that supporting the more nuanced characteristics of *rare high interaction degrees* and *scattered local* effects are essential for scaling sharing-based approaches to large configuration spaces.

We summarize which properties are supported by each of the discussed tools in Table 3. Currently, the tools that exploit more characteristics are also based on a more heavy-weight infrastructure (i.e., higher engineering effort for the analysis and higher runtime effort to execute individual instructions). We hope that our analysis infrastructure helps to identify a sweet spot for exploiting the most relevant interaction characteristics, without the overhead of our current dynamic analysis implementation in VarexJ.

In several traces, we measured interactions of which not all might be intended. We conjecture that our dynamic analysis might be useful for developers to understand the sources of interactions and to build maintainable and assurable software.

7. RELATED WORK

Characteristics of Interactions. Despite much research on highly-configurable systems, the nature of configuration-related interactions is not well understood, especially at the code level. In studying bug reports, many studies found that the majority of reported configuration-related bugs are caused by individual options or interactions among only few options with only few defects at higher degrees [1,17,21,26,42, 47, 53]; but none of these studies is based on a configuration-complete analysis. Manual search for feature interactions in requirements in telecommunications and electronic mail has focused primarily on pairwise interactions [29, 40]. The few studies that systematically analyzed entire configuration spaces found also interactions among more options, such as a linker fault in Busybox that involved 11 options [35]. Where prior work primarily focused on the degree of interaction faults, we define and monitor measures for interaction degrees and interaction overhead to assess configuration complexity of both data and control flow interactions.

Closest to our analysis of interactions at the execution level, Reisner et al. used symbolic execution to explore different paths of test cases in three C programs (9–14KLOC, 13–30 options) and found interactions among 7 of 30 options in one system [55]. Executing configurations separately, they measured the effect of interactions on control flow only (with the goal of increasing test coverage), whereas we specifically monitor the effect of interactions on data to measure configuration complexity (to assess whether a configuration-complete approach is feasible), especially regarding the different notions of local variability, e.g., using benchmarks B3–B5 and the complexity measure of interaction overhead.

Analyses for Configurable Systems. As already discussed in Sec. 2, analysis strategies for configuration spaces are usually based on sampling or sharing. Combinatorial interaction testing is a state-of-the-art sampling strategy to guarantee coverage of all combinations among τ configuration options [17, 21, 53]. Researchers have explored many other sampling strategies for configuration spaces, using machine learning [58] or optimizing code coverage [60], that all exploit similar assumptions of irrelevant or orthogonal variability.

The software-product-line community has extensively investigated strategies to analyze large configuration spaces through sharing [61]. Although the sharing techniques differ in details, they often follow similar ideas of analyzing the entire system at once, splitting where necessary and joining at fine granularity. Many techniques have been explored for efficient representation and reasoning about (partial) configuration spaces [25, 50, 64]. Our dynamic analysis, based on variability-aware execution, adopts many of those insights from static analyses to dynamic execution of programs.

Several researchers have recently explored forms of variability-aware execution that aggressively exploit sharing during the execution of alternatives of a program, particularly exploiting locality of options in control flow and data by storing variations locally with choices. Variability-aware interpreters for the WHILE language [36] and a subset of PHP [51] have demonstrated significant possible speedups over a brute-force approach. Kim's shared execution, also based on JAVAPATHFINDER, exploits sharing within functions, though not across function boundaries, evaluated on small-scale Java programs [38] (unfortunately the tool is not available, so we could not compare it in our evaluation). Austin and collaborators have explored the same concepts (named faceted values in their work) for JavaScript and a subset of Python, framed in the context of secure information flow analysis [6, 7]. Our dynamic analysis in VarexJ is built on the same ideas and is the first variability-aware interpreter that supports a full mainstream programming language and can analyze existing programs of significant size.

Some researchers have additionally explored static strategies to identify the scope of options and other changes as well as their potential interactions, based on slicing or data-flow analysis [3, 12, 44]. While such analyses can identify potential interactions without the need of specific inputs, static analyses are conservative and tend to significantly over approximate potential interactions. Instead of detecting interactions, some approaches aim to identify the cause of a configuration fault once it occurs [66, 69] or automatically resolve interactions with a default strategy [10, 33]. Such approaches are orthogonal to our analysis.

Analyses Beyond Configurable Systems. Addressing exponentially growing search spaces through some form of sharing is common for analyses in many domains. For example, even in finite models, the search space in model checking often exceeds the available memory, known as the state-space-explosion problem [8,9]. Symbolic model checking approaches increase sharing by representing states more compactly with symbolic techniques [16, 20]. Other techniques have been explored to further reduce redundancies in state representations, such as separation of frequently changing values [4, 63], and compact encoding and manipulation of multiple states [22,57]. Our results can indicate the expected effectiveness of each sharing strategy when applied to configuration spaces, based on the expected characteristics of interactions, as discussed in Sec. 4 and 6.

Our dynamic analysis monitors differences among multiple executions similar to concepts of *multi execution* [23, 24]. Most multi-execution approaches execute variants separately and use external synchronization mechanisms [32,46], though some approaches explicitly share redundancies, but only for a small number of variants (usually two) [59, 62]. In contrast, we explicitly monitor interactions among multiple options in an exponential configuration space.

8. CONCLUSION

Undesired interactions challenge quality assurance for highly-configurable software, as they are typically unknown and can result in faults and security vulnerabilities. Their detection is a challenge as the configuration space of such systems grows up to exponentially in the number of configuration options. Existing analyses try to scale with assumptions about interactions. However, whether these assumptions are valid and how much we can speed up analyses in future is not well understood. With VAREXJ, we implemented a dynamic analysis for Java to quantify different characteristics of interactions with benchmarks and to analyze real-world programs. We found that essential configuration complexity induced by real-world interactions is usually low, making configuration-complete analyses feasible. Based on our insights, we discussed typical characteristics of interactions, which can be exploited by future approaches for quality assurance of configurable systems.

Acknowledgments. This work has been supported in part by the BMBF grant (01IS14017A, 01IS14017B), the NSF awards 1318808 and 1552944, the Science of Security Lablet (H9823014C0140), AFRL and DARPA (FA8750-16-2-0042), and the U.S. Department of Defense through the SERC (H9823008D0171).

9. REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *ASE*, pages 421–432. ACM, 2014.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. In *TACAS*, pages 134–138. Springer, 2007.

[3] F. Angerer, A. Grimmer, H. Prähofer, and P. Grünbacher. Configuration-Aware Change Impact Analysis. In *ASE*, pages 385–395. IEEE, 2015.

[4] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein. Domain Types: Abstract-Domain Selection Based on Variable Usage. In *HVC*, pages 262–278. Springer, 2013.

[5] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *ICSE*, pages 482–491. IEEE, 2013.

[6] T. H. Austin and C. Flanagan. Multiple Facets for Dynamic Information Flow. In *POPL*, pages 165–178. ACM, 2012.

[7] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted Execution of Policy-Agnostic Programs. In *PLAS*, pages 15–26. ACM, 2013.

[8] T. Ball, V. Levin, and S. K. Rajamani. A Decade of Software Model Checking with SLAM. *Comm. ACM*, 54(7):68–76, 2011.

[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *STTT*, 9(5):505–525, 2007.

[10] C. Bocovich and J. M. Atlee. Variable-Specific Resolutions for Feature Interactions. In *FSE*, pages 553–563. ACM, 2014.

[11] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *PLDI*, pages 355–364. ACM, 2013.

[12] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression Tests to Expose Change Interaction Errors. In *ESEC/FSE*, pages 334–344. ACM, 2013.

[13] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *AOSD*, pages 13–24. ACM, 2012.

[14] F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[15] G. Bruns. Foundations for Features. In *Feature Interactions in Telecommunications and Software Systems VIII*, pages 3–11. IOS Press, 2005.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, pages 428–439. IEEE, 1990.

[17] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the Testing and Testability of Software Product Lines. In *SPLC*, pages 241–255. Springer, 2010.

[18] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

[19] I. D. Carmo Machado, J. D. McGregor, Y. a. C. Cavalcanti, and E. S. De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *IST*, 56(10):1183–1199, 2014.

[20] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *ICSE*, pages 321–330. ACM, 2011.

[21] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *ISSTA*, pages 129–139. ACM, 2007.

[22] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. *TSE*, 34(5):597–613, 2008.

[23] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *CCS*, pages 748–759. ACM, 2012.

[24] D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *SP*, pages 109–124. IEEE, 2010.

[25] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *TOSEM*, 21(1):6:1–6:27, 2011.

[26] B. J. Garvin and M. B. Cohen. Feature Interaction Faults Revisited: An Exploratory Study. In *ISSRE*, pages 90–99. IEEE, 2011.

[27] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *CCS*, pages 38–49. ACM, 2012.

[28] M. Greiler, A. v. Deursen, and M.-A. Storey. Test Confessions: A Study of Testing Practices for Plug-in Systems. In *ICSE*, pages 244–254. IEEE, 2012.

[29] R. J. Hall. Fundamental Nonmodularity in Electronic Mail. *ASE*, 12(1):41–79, 2005.

[30] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[31] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI*, pages 453–464. ACM, 2009.

[32] P. Hosek and C. Cadar. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *ASPLOS*, pages 339–353, 2015.

[33] M. Jackson and P. Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *TSE*, 24(10):831, 1998.

[34] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *TOSEM*, 21(3):14:1–14:39, 2012.

[35] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *OOPSLA*, pages 773–792. ACM, 2012.

[36] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *FOSD*, pages 1–8. ACM, 2012.

[37] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *AOSD*, pages 57—68. ACM, 2011.

[38] C. H. P. Kim, S. Khurshid, and D. Batory. Shared Execution for Efficiently Testing Product Lines. In *ISSRE*, pages 221–230. IEEE, 2012.

[39] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat:

Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *ESEC/FSE*, pages 257–267. ACM, 2013.

[40] M. Kolberg, E. Magill, D. Marples, and S. Reiff. *Feature Interactions in Telecommunication Systems VI*, chapter Results of the Second Feature Interaction Contest, pages 311–325. IOS Press, 2000.

[41] J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: An Integrated Approach to Distributed Computer Control Systems. *IEE Proc. Computers and Digital Techniques*, 130(1):1–, 1983.

[42] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software Fault Interactions and Implications for Software Testing. *TSE*, 30:418–421, 2004.

[43] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *ESEC/FSE*, pages 81–91. ACM, 2013.

[44] M. Lillack, C. Kästner, and E. Bodden. Tracking Load-Time Configuration Options. In *ASE*, pages 445–456. ACM, 2014.

[45] R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE*, pages 10–24. Springer, 2001.

[46] M. Maurer and D. Brumley. Tachyon: Tandem Execution for Efficient Live Patch Testing. In *USENIX Security Symposium*, pages 617–630, 2012.

[47] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *ICSE*, pages 664–675. ACM, 2016.

[48] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *ECOOP*, volume 37 of *LIPIcs*, pages 495–518. Schloss Dagstuhl–LZI, 2015.

[49] J. Meinicke. VarexJ: A Variability-Aware Interpreter for Java Applications. Master's thesis, University of Magdeburg, 2014.

[50] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *SPLC*, pages 231–240. Software Engineering Institute, 2009.

[51] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *ICSE*, pages 907–918. ACM, 2014.

[52] A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature Interaction: The Security Threat from within Software Systems. *Progress in Informatics*, pages 75–89, 2008.

[53] C. Nie and H. Leung. A Survey of Combinatorial Testing. *CSUR*, 43(2):11:1–11:29, 2011.

[54] M. Plath and M. Ryan. Feature Integration Using a Feature Construct. *SCP*, 41(1):53–84, 2001.

[55] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *ICSE*, pages 445–454. ACM, 2010.

[56] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. QuEval: Beyond High-Dimensional Indexing à la Carte. In *VLDB*, pages 1654–1665. VLDB Endowment, 2013.

[57] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *FSE*, pages 842–853. ACM, 2015.

[58] C. Song, A. Porter, and J. S. Foster. iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees. *SE*, 40(3):251–265, 2014.

[59] W. N. Sumner, T. Bao, X. Zhang, and S. Prabhakar. Coalescing Executions for Fast Uncertainty Analysis. In *ICSE*, pages 581–590. ACM, 2011.

[60] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static Analysis of Variability in System Software: The 90,000# Ifdefs Issue. In *Proc. USENIX*, pages 421–432, 2014.

[61] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 47(1):6:1–6:45, 2014.

[62] J. Tucek, W. Xiong, and Y. Zhou. Efficient Online Validation with Delta Execution. *SIGARCH*, 37(1):193–204, 2009.

[63] A. von Rhein, S. Apel, and F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *JPF Workshop*, 2011.

[64] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Onward!*, pages 213–226. ACM, 2014.

[65] B. Xin, W. N. Sumner, and X. Zhang. Efficient Program Execution Indexing. In *PLDI*, pages 238–248. ACM, 2008.

[66] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do Not Blame Users for Misconfigurations. In *Proc. Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.

[67] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *SOSP*, pages 159–172. ACM, 2011.

[68] P. Zave. *Software Requirements and Design: The Work of Michael Jackson*, chapter Modularity in Distributed Feature Composition, pages 267–290. Good Friends Publishing Company, 2009.

[69] S. Zhang and M. D. Ernst. Automated Diagnosis of Software Configuration Errors. In *ICSE*, pages 312–321. IEEE, 2013.

[70] X. Zhang and R. Gupta. Whole Execution Traces and Their Applications. *ACM Trans. Archit. Code Optim.*, 2(3):301–334, 2005.