Blackbox Observability of Features and Feature Interactions

Kallistos Weis
Saarland University
Germany

Leopoldo Teixeira
Federal University of Pernambuco
Brazil

Clemens Dubslaff
Eindhoven University of Technology
The Netherlands

Sven Apel
Saarland University
Germany

ABSTRACT

Configurable software systems offer user-selectable features to tailor them to the target hardware and user requirements. It is almost a rule that, as the number of features increases over time, unintended and inadvertent *feature interactions* arise. Despite numerous definitions of feature interactions and methods for detecting them, there is no procedure for determining *whether the effect of a feature interaction could be, in principle, observed* from an external perspective. In this paper, we devise a *decision procedure* to verify whether the effect of a given feature or potential feature interaction could be isolated by *blackbox observations* of a set of system configurations. For this purpose, we introduce the notion of *blackbox observability*, which is based on recent work on *counterfactual reasoning* on configuration decisions. *Direct* observability requires a single reference configuration to isolate the effect in question, while the broader notion of *general* observability relaxes this precondition and suffices with a set of reference configurations. We report on a series of experiments on community benchmarks as well as real-world configuration spaces and models. We found that (1) deciding observability is indeed tractable in real-world settings, (2) constraints in real-world configuration spaces frequently limit observability, and (3) blackbox performance models often include effects that are *de facto* not observable.

1 INTRODUCTION

The question of *observability*, that is, which properties of a system can be determined from external observations, is fundamental in various disciplines, including philosophy [18], applied physics [39], control theory [56], and runtime monitoring [37]. Given a set of observations of a system's behavior, it is possible to infer information about the system's internal state and properties. The most informative behaviors are those that expose different effects along with different observations, enabling counterfactual reasoning and allowing conclusions about system internals from external observations [62]. The decision problem of *observability* refers to the question of whether it is, *in principle*, possible to make such property-discriminating observations. This problem is of utter importance for system analysis and design: If a system property is, in principle, observable, engineers can collect and analyze a proper set of observations for which the system exhibits different properties. For example, testing a system's performance would involve a set of test cases that trigger both high and low performance behavior. Conversely, if a system property is, in principle, *not* observable, all analyses of observations will lack a factual basis, and there is no chance to ever find a set of observations that expose this property.

A premise of our work is that the observability problem is fundamental in designing and analyzing configurable software systems. A *configurable software system* provides a set of features (e.g., configuration options) that a user can select to tailor it to the target hardware and user requirements. In fact, most non-trival software systems today are configurable [2]. The combinatorics of selecting features typically leads to a huge number of possible *system configurations* [4]. The behavior and properties of a system greatly depend on its configuration. In particular, interactions among features can lead to undesired and inadvertent behaviors, which is known as the *feature-interaction problem* [1, 6, 45]. The crux is that, due to the often huge number of system configurations, it is infeasible or even impossible to test all system configurations covering all potential feature interactions [1, 6, 26, 45].

A further complication is that there are typically *constraints* among features that must be satisfied for them to be selectable (e.g., a feature requires another), rendering only the system configurations as *valid* that fulfill such constraints [4]. For an invalid configuration, the corresponding software cannot be deployed in practice and thus can never provide observations for analysis. The observability of the effects of individual features and interacting features is hence *fundamentally limited* by the constraints imposed on valid configurations. For example, consider a database system in which an authentication feature requires an encryption feature to be enabled. In this case, it is impossible to observe any interaction between the authentication feature and any other feature of the system in a blackbox manner. While such observability questions naturally arise [15, 46, 65], we are not aware of any foundational method to deciding observability in configurable systems.

In this paper, we address the fundamental question of whether and how effects arising from individual features and feature interactions can be, in principle, observed without internal knowledge of the system—knowledge that might be unavailable or hard to obtain. Our goal is to establish a *decision procedure* that, given the constraints on valid configurations, decides whether an effect could be observed by running and comparing observations of a set of system configurations (e.g., by comparing the runtime of two system configurations that differ only in the selection of one feature). As

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE 2024, October 2024, Sacramento, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

a foundation, we formally introduce *direct* observability and then extend it to the broader notion of *general* observability. While *direct* observability requires a single configuration as a reference to render the effect in question observable, *general* observability relaxes this precondition and suffices with a set of system configurations as a reference. We then devise a decision procedure implementing our formal definitions of direct and general observability. To evaluate our decision procedure, we conducted a series of experiments on community benchmarks and real-world configuration spaces and models to assess the feasibility of deciding observability in practice and to investigate the impact of configuration constraints on observability. We found that (1) deciding blackbox observability is indeed feasible in real-world settings, (2) constraints in real-world configuration spaces frequently limit blackbox observability, and (3) a popular class of models (blackbox performance models [32]) often include effects that are *de facto* not blackbox observable, which limits their interpretability.

Contributions. This paper makes the following contributions:

- We describe the problem of observability of features and feature interactions in configurable systems.
- We introduce formal definitions of direct and general observability of features and feature interactions in configurable systems.
- We devise a decision procedure that decides observability.
- We evaluate the procedure on community benchmarks and real-world configuration spaces and models.

Supplement. The implementation of our decision procedure, the data, and the code to run the experiments are publicly available.[1]

2 BACKGROUND

In this section, we provide the necessary background on configurable systems that is used throughout this paper. We explain the abstract view on configurable systems that we use, as well as the necessary notions to express and reason about variability in the presence of constraints. We also discuss the notion of blackbox performance models, which are used to model the performance of configurable software systems.

Configurable Systems. A common approach to model the variability of a software system is to use features [33] (i.e., separate units of functionality). We denote the set of all features with $\mathcal{F}$. Features can be selected or deselected, which means that the corresponding functionality of the software system is included in the system or not, respectively. We abstract this behavior by treating features as Boolean variables, where an included functionality (selected feature) is represented by $\top$, and an excluded functionality (deselected feature) is represented by $\bot$. Assigning a truth value (i.e., $\top$ or $\bot$) to each feature of the software system is called a *configuration.* We use $\mathcal{V}$ to denote the *valid configuration space*, that is, the set of all valid configurations of a system, which is encoded by a feature model.

Example 2.1. Consider a simple E-MAIL system over the set of features $\mathcal{F} = \{m, s, e, c, a, r\}$, defining the base functionality of the email system, as well as optional functionalities for signing and encrypting with a specific encryption algorithm, Caesar, AES, or RSA. If the Caesar encryption algorithm is not selected, then the signature feature must be selected. Figure 1 shows a feature diagram representing these features and constraints, which results in the set of valid configurations

$$\mathcal{V} = \{mec, ms, msec, msea, mser\}.$$

Note that, to simplify the notation, we write a configuration as a string of features, where a feature is included if it is present in the string, and excluded otherwise.

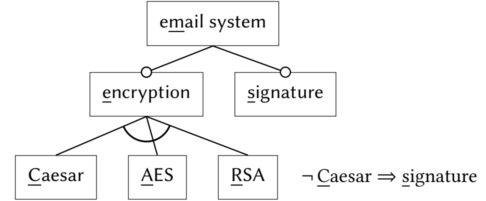

Figure 1: Feature model of a small E-MAIL system

[1] https://github.com/BlackboxObservability/UatuEvaluation/

A *partial configuration* is a configuration that assigns a truth value to only a subset of features. We use $\Delta(\mathcal{F})$ to denote the set of all partial configurations over $\mathcal{F}$, and $\Theta(\mathcal{F})$ for the set of all configurations. We denote by $\mathsf{supp}(\partial)$ the set of all features $f \in \mathcal{F}$ that are assigned a value by a partial configuration ∂. Given a configuration η and a set of features $X \subseteq \mathcal{F}$, we denote by $\mathsf{switch}(\eta, X)$ a configuration that is derived by switching the values of all features in X (i.e., $\mathsf{switch}(\eta, X)(f) = \neg\eta(f)$ for all $f \in X$).

The semantics of a partial configuration $[\![\partial]\!]$ is defined as the set of all configurations that satisfy the constraints of ∂. We refer to all configurations that satisfy the constraints of a partial configuration ∂ (that is $\eta \in [\![\partial]\!]$) as configurations that are *consistent* with the partial configuration ∂. Expanding a partial configuration ∂ w.r.t. a set of features $X \subseteq \mathcal{F}$ means to keep the constraints for all features not contained in X and remove the constraints for all features in X. This operation is denoted by $\partial\uparrow_X$, where $\mathsf{supp}(\partial\uparrow_X) = \mathsf{supp}(\partial) \setminus X$ and $\partial\uparrow_X(f) = \partial(f)$ for all $f \in \mathsf{supp}(\partial) \setminus X$.

Example 2.2. Consider the E-MAIL system from Example 2.1. Suppose a user is interested in configurations that include the signature feature. We can express this by the partial configuration $\partial = \{s \mapsto \top\}$. For this partial configuration, the set of valid configurations included in $[\![\partial]\!]$ is $\mathcal{V} \cap [\![\partial]\!] = \{ms, msec, msea, mser\}$ and the support is $\mathsf{supp}(\partial) = \{s\}$. In this example, the expansion of the partial configuration ∂ (i.e., $\partial\uparrow_X$ with $X = \{s\}$) means that the semantics of the partial configuration is the set of all configurations (i.e., $[\![\partial\uparrow_X]\!] = \Theta(\mathcal{F})$).

Blackbox Performance Models. Performance (e.g., run time, latency, throughout, energy consumption) is a key property of many software systems. Modeling and predicting the performance of a software system is already challenging, but the fact that a system may come in many variants (i.e., configurations) considerably complicates this task. In response to this challenge, researchers have devised a range of performance modeling techniques that allow a user to predict the performance of a configuration depending on the selected features [25, 30, 52]. While there is a multitude of types of models [30], a canonical representation of a performance model is

a function, in additive form, that assigns performance influences to each feature and feature interaction that is relevant for the performance behavior. The performance influences are typically learned from a set of reference configurations, which are configurations that have been executed and measured [30].

Example 2.3. In our E-MAIL system a performance model could be as follows:

$$\Pi = 5 \cdot m + 10 \cdot s + 15 \cdot e + 10 \cdot s \cdot e + \dots .$$

The variables in this model are the system features, and the coefficients are the performance influences of the corresponding features (single variables, e.g., s) and feature interactions (multiplicative terms, e.g., $s \cdot e$). The performance of a configuration is the sum of the performance influences of features and interactions that are selected in such configuration ($\top$ is mapped to 1 and $\bot$ is mapped to 0). The partial configuration that corresponds to the interaction between the features s and e is $\partial_{se} = \{s \mapsto \top, e \mapsto \top\}$.

It is important to note that these models have also been used for explanation [34, 40, 41, 52, 60], besides prediction. For example, to identify the feature that has the largest influence on performance (e in our example). However, the learning procedure is typically optimized for prediction accuracy, instead of explainability, which may lead to wrong conclusions, as we discuss in Section 5.

3 BLACKBOX OBSERVABILITY

Observability is a well-studied concept in a wide variety of research areas, such as philosophy, applied physics, control theory, and runtime monitoring [18, 37, 39, 56]. In this paper, we address the fundamental question of whether effects (e.g., correctness, reliability, or runtime) arising from individual features or feature interactions can be observed without internal knowledge of the system. This understanding can help us judge the interpretability of models that use those features and feature interactions to explicate the behavior of the system (e.g., stability, safety, or energy consumption).

In what follows, we introduce the notion of blackbox observability for individual features and feature interactions (i.e., partial configurations) in the context of configurable systems. We start with a formal definition of *direct observability* and lift it to *general observability*, which is more general, as it combines direct and *indirect* observability. It is important to note that our definitions of observability are concerned with the structural constraints between the features of the configurable system. As such, our definitions only make statements about the nature of the constraints of the configuration space, not about the actual behavior of the system and its features. This includes masking or shielding effects, which might prevent the observation of the effect of certain features or feature interactions when executing the system, even though they might be generally observable.

Direct Observability. The basic reasoning task to decide whether a partial configuration is directly observable is to *check whether there is a witness and a counter witness*[2] *of the partial configuration.* A witness of a partial configuration is a configuration that is consistent with the partial configuration and a counter witness is a valid configuration that is not consistent with any feature assignment of the partial configuration. A partial configuration is called directly observable if there is a configuration that fulfills all constraints of the partial configuration and a configuration that does not fulfill any constraints of the partial configuration, but shares all feature assignments of the first configuration except for the features in the support of the partial configuration.

Example 3.1 (Direct observability). Continuing with our E-MAIL example from Example 2.1, let us consider the partial configuration $\partial_{er} = \{e \mapsto \top, r \mapsto \top\}$ with its support S ($= \mathsf{supp}(\partial_{er})) = \{e, r\}$. We are interested in whether or not ∂_{er} is directly observable, for example, to learn whether it is possible to detect a 2-wise feature interaction between e and r via a blackbox analysis. There is only one candidate $\bar{\eta}$ in the set of valid configurations that is not included in $[\![\partial_{er}]\!]$, for a counter witness of ∂_{er}: $\bar{\eta} = ms$. As we can see, switching all features in the support of ∂_{er} (i.e., $S = \{e, r\}$) in $\bar{\eta}$ (i.e., $\mathsf{switch}(\bar{\eta}, S) = mser$) leads to a witness of ∂_{er} (i.e., $mser$). Since $mser$ is a valid configuration (i.e., $mser \in \mathcal{V}$), it is indeed a witness of ∂_{er}. This leads to the conclusion that ∂_{er} is directly observable. In contrast, if we only switch e in $\bar{\eta}$ (i.e., $\mathsf{switch}(\bar{\eta}, \{e\}) = mse$), we would not obtain a witness of ∂_{er}, because mse is not a valid configuration (i.e., $mse \notin \mathcal{V}$) and, more importantly, by comparing mse with ms, we would not be able to observe the effect of r.

Intuitively, to decide observability of a partial configuration ∂, we have to identify one configuration c_1 that is consistent with the partial configuration ∂ and a corresponding configuration c_2 that agrees in all feature assignments with c_1 except for the features in the support of ∂. For all features in the support of ∂, c_2 has to disagree with c_1. Then, c_2 is a counter witness of ∂, c_1 is a witness of ∂, and ∂ is directly observable.

Inspired by the concept of counterfactual reasoning [17, 38], we define direct observability as follows:

Definition 3.2 (Direct observability). Let $\partial \in \Delta(\mathcal{F})$ be a partial configuration where $S = \mathsf{supp}(\partial)$. Then, ∂ is *directly observable* if there is a configuration $\bar{\eta} \in \mathcal{V} \setminus [\![\partial]\!]$ with $\mathsf{switch}(\bar{\eta}, S) \in \mathcal{V} \cap [\![\partial]\!]$.

In Definition 3.2, we require that there is a configuration $\bar{\eta}$ (i.e., a counter witness) of the partial configuration ∂ that is a partial inverse[3] of a configuration $\mathsf{switch}(\bar{\eta}, S)$ (i.e., a witness) of ∂. We require that S, the basis on which we search a counter witness, is the set of all features that are defined in ∂. Then, $\bar{\eta}$ is a counter witness of ∂, if there is a witness ($\mathsf{switch}(\bar{\eta}, S)$) that disagrees with all feature assignments of $\bar{\eta}$ in the support of ∂.

Example 3.3 (Computation of direct observability). Let us illustrate the decision about direct observability for the partial configuration from Example 3.1 by Algorithm 1. The input to Algorithm 1 is $\partial_{er} = \{e \mapsto \top, r \mapsto \top\}$. In Line 1, we check whether there is a configuration in $[\![\partial_{er}]\!]$ that is consistent with the constraints of the feature model. Then, in Line 2, we compute the support of ∂_{er}, which is $S = \{e, r\}$. In Line 4, we check whether there is a configuration η in $[\![\partial_{er}]\!]$ for which the configuration $\mathsf{switch}(\eta, S)$

[2] A counter witness $\bar{\eta}$ represents a valid configuration for which all features in the support of a partial configuration ∂ are switched in configuration η. The notion of *counterfactuals* is closely related but is concerned with a change of an *effect* induced by a change of the feature selection [17].

[3] By partial inverse, we refer to a configuration c_1 that agrees with another configuration c_2 in all feature selections but the feature selections in the support of a given partial configuration ∂.

Algorithm 1: Computation of direct observability

```
    input  : ∂ ∈ Δ(F)
    output : ⊤ if ∂ is directly observable, ⊥ otherwise
1   if ⟦∂⟧ ∩ V = ∅ then return ⊥
2   S := supp(∂)
3   forall η ∈ ⟦∂⟧ ∩ V do
4   |   if switch(η, S) ∈ V then return ⊤
5   return ⊥
```

Algorithm 2: Computation of general observability

```
    input  : ∂ ∈ Δ(F)
    output : ⊤ if ∂ is generally observable, ⊥ otherwise
1   forall δ ∈ partitions(∂)              ▷ all possible partitions of ∂[5]
2   do
3   |   ω := ⊤
4   |   forall ∂′ ∈ δ do
5   |   |   if directly-observable(∂′) = ⊥ then ω := ⊥
6   |   if ω = ⊤ then return ⊤
7   return ⊥
```

is also a valid configuration (i.e., there is a counter witness). In our example, there is such a configuration pair (i.e., $mser \in \mathcal{V}$ and $ms \in \mathcal{V}$), which leads to the conclusion that ∂_{er} is directly observable.

General Observability. In Example 3.1, we have seen that the partial configuration ∂_{er} is directly observable. However, there is not always a counter witness $\bar{\eta}$ witness η pair for a partial configuration ∂. Even in the presence of simple constraints among features, such as implications, it is not always possible to directly observe arbitrary partial configurations in a system. Consider as a small example the constraint between the Caesar algorithm and the signing feature from our E-MAIL example. Because of this constraint, it is not possible to directly observe the partial configuration $\partial_{sc} = \{s \mapsto \top, c \mapsto \top\}$, as there is no partial inverse for the only valid configuration in $[\![\partial_{sc}]\!]$ (i.e., $msec$) and $S = \{s, c\}$.

However, in certain cases we can *indirectly* observe partial configurations if it is possible to split the support of the partial configuration such that one can observe each part on its own.

Example 3.4. Consider the partial configuration $\partial_{sc} = \{s \mapsto \top, c \mapsto \top\}$ from our E-MAIL example. We saw that ∂_{sc} is not directly observable, as there is no counter witness $\bar{\eta}$ for the only valid configuration $msec$ in $[\![\partial_{sc}]\!]$. However, assume that we are able to observe the partial configurations $\partial_s = \{s \mapsto \top\}$ and $\partial_c = \{c \mapsto \top\}$. We would then be able to indirectly observe ∂_{sc} by splitting the support of ∂_{sc} into $S_1 = \{s\}$ and $S_2 = \{c\}$.

Intuitively, assume we have a partial configuration ∂, that is not directly observable, but we are able to split the support of ∂ into smaller partial configurations ∂_i such that we are able to observe *each* ∂_i by its own. Then, we can *indirectly* observe ∂. This leads to the definition of *general observability*[4], for which we recall that a partition of a set S is a set $\{S_0, S_1, \ldots, S_n\}$ for some $n \leq |S|$ such that $S = \bigcup_{0 \leq i \leq n} S_i$, $S_i \neq \emptyset$, and $S_i \cap S_j = \emptyset$ for all $i \neq j \in [0, n]$.

Definition 3.5 (General observability). Let $\partial \in \Delta(\mathcal{F})$ be a partial configuration with support $S = \mathsf{supp}(\partial)$. Then, ∂ is *generally observable* if there is a partition $\{S_0, S_1, \ldots, S_n\}$ of S, such that for all $k \in [0, n]$ there is a configuration $\bar{\eta}_k \in \mathcal{V} \setminus [\![\partial \uparrow_{S \setminus S_k}]\!]$ where $\mathsf{switch}(\bar{\eta}_k, S \setminus S_k) \in \mathcal{V} \cap [\![\partial \uparrow_{S \setminus S_k}]\!]$.

Note that if ∂ is directly observable, then it is also generally observable, witnessed by the trivial partition $\{S\}$ and the fact that $\partial \uparrow_{S \setminus S} = \partial$. The main difference between direct observability and general observability is the coverage criterion over the support of the partial configuration ∂. Direct observability requires that there is a counter witness $\bar{\eta}$ of ∂ that is a partial inverse of a witness in $[\![\partial]\!]$. In contrast, general observability relaxes this constraint by introducing a partition S_i with $0 \leq i \leq n$ of the support of ∂.

[4]General observability combines direct and indirect observability

Example 3.6 (General observability). Let us illustrate the notion of general observability using our E-MAIL example from Figure 1 and the partial configuration $\partial_{sec} = \{s \mapsto \top, e \mapsto \top, c \mapsto \top\}$. The constraints in our example enforce that the signature feature has to be enabled whenever the Caesar algorithm is disabled. This leads to the conclusion that ∂_{sec} is not directly observable, as there is no counter witness $\bar{\eta}$ of ∂_{sec} for a witness in $[\![\partial_{sec}]\!]$ (i.e., $m \notin \mathcal{V}$).

However, let us consider the sets $S_1, S_2 \subseteq S = \mathsf{supp}(\partial_{sec})$ with $S_1 = \{s\}$ and $S_2 = \{e, c\}$ ($S \setminus S_1 = \{e, c\}$ and $S \setminus S_2 = \{s\}$, respectively). The partition of S into the two subsets S_1 and S_2 corresponds to one δ in the set of all possible partitions of ∂_{sec} (see Algorithm 2, Line 1). Consider the two configurations $\bar{\eta}_1 = mec$ and $\bar{\eta}_2 = ms$.

As we can see,

$$\mathsf{switch}(\bar{\eta}_1, S_1) = msec \in \mathcal{V} \cap [\![\partial_{sec} \uparrow_{S \setminus S_1}]\!]$$
$$\mathsf{switch}(\bar{\eta}_2, S_2) = msec \in \mathcal{V} \cap [\![\partial_{sec} \uparrow_{S \setminus S_2}]\!].$$

This means that all partial configurations in this partition are directly observable (see Algorithm 2, Line 4) and, therefore, ∂_{sec} is generally observable (see Algorithm 2, Line 6). That is, we are able to indirectly observe ∂_{sec} by splitting the support of ∂_{sec} into S_1 and S_2. By doing so, we can reason about a potential 3-wise feature interaction between s, e, and c using only blackbox observations.

4 EXPERIMENT SETUP

To evaluate our notion and decision procedure of blackbox observability, we conduct an empirical study on a set of community benchmarks and real-world configurable software systems.

4.1 Research Questions

Our evaluation addresses three research questions: We want to learn whether our notion of observability is effectively computable, understand the impact of constraints on observability in practical settings, and investigate the effect of observability on the correctness of blackbox performance models learned in previous work.

Effectivity of Observability Computation. Computing the observability of partial configurations for a configurable software system is a combinatorial problem. The number of partial configurations of a system grows exponentially with the number of

[5]A partition of a partial configuration ∂ w.r.t. to its support $S = \mathsf{supp}(\partial)$ is the set of partial configurations $\partial_1 \ldots \partial_n$ constructed via the partition of S (i.e., $S_1, \ldots, S_n$) by $\partial_1 = \partial \uparrow_{S_1}, \ldots, \partial_n = \partial \uparrow_{S_n}$.

features. Therefore, we want to learn whether our notion of observability is effectively computable in a practical setting. By effectively computable, we mean that the procedure for computing observability for all partial configurations of a specific size is applicable to community benchmarks and real-world systems.

$\mathbf{RQ}_1$: Can we effectively compute the observability of partial configurations?

Observability of Partial Configurations. One characteristic of configurable software systems is the multitude of constraints among features, meaning that the set of valid configurations is much smaller than the set of all feature selections [43]. We investigate the impact of these constraints on the presence and prevalence of observable partial configurations. This is especially important for state-of-the-art blackbox analysis methods, which rely on the assumption that all partial configurations can be used to explain the behavior of the configurable software system. Therefore, we want to learn how many partial configurations are observable in our community benchmarks and real-world subject systems to evaluate the impact of the constraints on the set of observable partial configurations. This provides insights into the applicability of blackbox methods for explaining the behavior of partial configurations in configurable software systems.

$\mathbf{RQ}_2$: What fraction of partial configurations is observable in community benchmarks and real-world subject systems?

Observability of Feature Interactions. Existing blackbox definitions of feature interactions inherently assume that a partial configuration representing a feature interaction is observable. For example, blackbox performance modeling methods have been successfully used to derive performance models that calculate the performance of a given configuration based on the contributions of individual features and feature interactions (see Section 2). The problem is that such performance models are typically trained by regression, optimizing for prediction accuracy rather than interpretability – the ability to explain the behavior of the system. So, while these models are used to *explain* the system behavior [34, 40, 41, 52, 60], it is actually not clear whether the explanations (i.e., influences of features and feature interactions) are correct (cf. Section 2). To assess the validity of the explanations, we determine how many feature interactions are observable in state-of-the-art blackbox performance models learned and interpreted in previous work. In contrast to the more general question about the observability of partial configurations in $\mathbf{RQ}_2$, we want to learn whether the features and feature interactions included in state-of-the-art performance-influence models are actually observable.

$\mathbf{RQ}_3$: What fraction of features and feature interactions in state-of-the-art blackbox performance models is observable?

4.2 Operationalization

To answer our research questions, we conduct an empirical study on a set of community benchmarks and real-world subject systems that we collected from the literature (see Section 4.4 for details). Based on the subject systems' feature models, we compute the sets of all partial configurations of a specific size and compute the observability of these partial configurations. We limit our attention to partial configurations of size one, two, and three, to keep the effort for experimentation feasible. To answer $\mathbf{RQ}_1$, we measure the overall time needed to compute the observability of all partial configurations. To answer $\mathbf{RQ}_2$, we count the number of observable partial configurations and relate this number to the total number of partial configurations of the considered sizes. To answer $\mathbf{RQ}_3$, we compute the percentage of observable features and feature interactions in a selection of blackbox performance models from the literature (see Section 4.4).

4.3 Implementation

To answer our research questions, we have implemented UATU[6], a prototype to compute the observability of partial configurations based on the definitions and algorithms given in Section 3. The prototype is implemented in Python, relying on DD[7], a library for building and manipulating binary decision diagrams. In particular, we use DD as an interface to the CUDD library [55].

UATU takes as input a feature model and a set of partial configurations, for which the observability shall be computed. The feature model is represented as a propositional formula in disjunctive normal form, where each variable in the formula represents a feature and each clause represents a valid configuration. First, we compute the set of valid configurations that are consistent with the partial configuration. Second, we compute the set of valid configurations that are candidates for a counter witness of the partial configuration. If there is a valid configuration in the second set that is a partial inverse of a valid configuration in the first set, then the partial configuration is observable.

Otherwise, the partial configuration is not directly observable, which means that there is no counter witness that is a partial inverse of a witness of the partial configuration. Next, we have to check whether there is a partition of the partial configuration into smaller partial configurations of which all parts are observable. If this is the case, then the partial configuration is generally observable; otherwise, it is not observable. Checking all possible partitions is computationally expensive. To reduce the computation time, we make use of lazy evaluation and memoization (i.e., we keep track of all previously computed observable and non-observable partial configurations).

4.4 Subject Systems

To evaluate our notion and decision procedure of observability, we have collected a diverse set of subject systems ranging from popular community benchmarks to real-world systems from the configurable systems community. We also included systems from recent work on blackbox performance modeling [32] to evaluate whether observability can be used to gain insight into their interpretability. A concise overview of the subject systems used to evaluate the *effectivity* of our notion as well as the *observability* of partial configurations is given in Table 1.

[6] UATU is one of the *Watchers* from Marvel Comics, an advanced species committed to observe the universe.
[7] The library can be found on https://github.com/tulip-control/dd.

Specifically, we selected the feature models of Minepump, Elevator, and CFDP from Cordy et al. [11], which are used to evaluate accompanying LTL properties on the modeled systems. From Rhein et al. [61], we selected the feature models of Apache, Curl, Email, h264, LinkedList, PKJab, Prevaylar, and ZipMe, which constitute a collection of real-world systems from the configurable systems community used for various experiments and case studies.

To include real-world subjects that have been used to evaluate and interpret blackbox performance models, we selected the models of two subject systems from different domains [31, 52], LLVM and Lrzip_1. This set of subject systems is complemented by the models of two subject systems from the literature on variability-aware probabilistic model checking: BSN [47] and VCL [13, 16].

In order to investigate the observability of partial configurations in state-of-the-art blackbox performance models, we selected 11 different models from a recent paper on blackbox performance modeling [32]. These models are based on the feature models of Brotli, FastDownward, HSQLDB, Lrzip_2, MariaDB, MySQL, OpenVPN, Opus, PostgreSQL, VP8, and Z3, which cover a huge variety of different domains and application areas.

5 RESULTS

In this section, we present the results of our experiments. First, we discuss statistics on observability in community benchmarks and real-world subject systems, including the computation time of our prototype implementation ($\mathbf{RQ}_1$) and the number of observable partial configurations of different sizes ($\mathbf{RQ}_2$). Second, we present the results of evaluating observability of features and their interactions in state-of-the-art blackbox performance models ($\mathbf{RQ}_3$).

5.1 Effectivity of Observability Computation

To evaluate the effectivity of our decision procedure for observability, we implemented a prototype, Uatu, based on the definitions given in Section 3. We used Uatu on 15 subject systems, for all possible partial configurations consisting of one, two, and three features. Table 1 summarizes the results of this evaluation. There, we see that the computation time to decide observability of partial configurations increases with the size of the partial configuration. While the total time needed to decide observability of partial configurations of size one (i.e., containing one feature) is less than a second for the majority of the subject systems, the total computation time for partial configurations of size two and three increases significantly for most systems (see Figure 2).

The total computation time for partial configurations of size two ranges from less than a second to several minutes, while the total computation time for partial configurations of size three ranges from a few seconds to several hours. In addition to the size of the partial configuration, the number of valid configurations and features in the subject system influences the computation time. For example, the computation time for partial configurations of size three in VCL is significantly lower than in Curl, even though the number of features in VCL is higher than in Curl. This is due to the fact that the number of constraints in the feature model of VCL is significantly lower than in Curl (i.e., the share of valid configurations in VCL is significantly higher than in Curl). There are two main reasons for the non-linear increase in computation

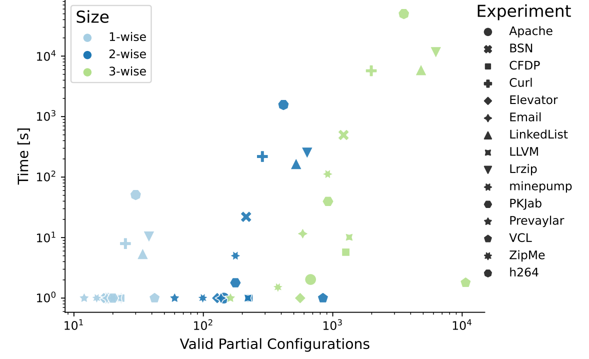

Figure 2: Total time required to compute the observability of all partial configurations (PCs) of a given size (color) for each subject system (marker shape). The x-axis shows the number of PCs of a given size for a system. To improve plot readability, we use a logarithmic scale for both axes.

time for partial configurations of increasing size: (1) the number of partial configurations of size n can be computed as $\frac{|\mathcal{F}|!}{(|\mathcal{F}|-n)!\cdot n!}$ and (2) the procedure has to check all possible partitions of a partial configuration into smaller partial configurations to decide general observability.

Summarizing our findings for $\mathbf{RQ}_1$, we conclude that observability is effectively computable. However, to keep the computation time feasible, one has to limit the size of the partial configurations.

5.2 Observability of Partial Configurations

Partial configurations are a powerful abstraction of internal variables to reason about the influence of features and their interactions on the external behavior of a system. However, configuration spaces of software systems are often highly constrained [31] (i.e., the number of valid configurations is significantly lower than the number of all possible configurations). This leads to the question of how many (or what fraction of) partial configurations are actually observable in a given subject system. In Table 1, we list the number of partial configurations ($|\mathcal{P}|$) that (1) have, at least, one valid representative ($|\mathcal{P}_{\mathcal{V}}|$), (2) are directly observable ($|\mathcal{D}_{O}|$), (3) are indirectly observable[8] ($|\mathcal{I}_{O}|$), and (4) are non-observable ($|\neg O|$) for partial configurations of size one, two, and three. For almost all systems, we see that the number of directly observable partial configurations of any size is lower than the number of partial configurations of size one that have, at least, one valid representative. For partial configurations of size one, we see that there are no generally observable partial configurations. This is by design, since it is not possible to partition a partial configuration of size one into smaller partial configurations.

Notably, while the number of indirectly observable partial configurations increases with the size of the partial configuration for half

[8] Note: To improve interpretability of the numbers, we report the number of generally observable partial configurations that are *not* directly observable, since all directly observable partial configurations are by definition generally observable.

Table 1: Statistics of general observability experiments

System	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{F}\rvert$	1-wise partial configurations						2-wise partial configurations						3-wise partial configurations					
			$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{P}_\mathcal{V}\rvert$	$\lvert\mathcal{D}_O\rvert$	$\lvert\mathcal{I}_O\rvert$	$\lvert\neg O\rvert$	Time	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{P}_\mathcal{V}\rvert$	$\lvert\mathcal{D}_O\rvert$	$\lvert\mathcal{I}_O\rvert$	$\lvert\neg O\rvert$	Time	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{P}_\mathcal{V}\rvert$	$\lvert\mathcal{D}_O\rvert$	$\lvert\mathcal{I}_O\rvert$	$\lvert\neg O\rvert$	Time
Apache	192	10	20	18	16	0	2	0.02	180	144	110	2	32	0.01	960	674	424	24	226	2.03
BSN	298	11	22	22	18	0	4	0.87	220	214	138	8	68	22.05	1 320	1 212	596	112	504	496.53
CFDP	56	13	26	23	20	0	3	0.02	312	225	134	46	45	0.23	2 288	1 261	386	574	301	5.74
Curl	768	14	28	25	16	0	9	7.96	364	286	118	0	168	218.82	2 912	1 985	544	0	1 441	5 721.79
Elevator	256	9	18	17	16	0	1	0.01	144	128	112	0	16	0.05	672	560	448	0	112	0.80
Email	40	10	20	18	8	0	10	0.05	180	137	28	0	109	0.66	960	586	60	4	522	11.60
LinkedList	204	19	38	34	6	0	28	5.42	684	521	38	2	481	166.50	7 752	4 809	144	36	4 629	5 939.70
LLVM	1 024	12	24	22	20	0	2	0.02	264	221	180	0	41	0.39	1 760	1 340	960	0	380	10.12
Lrzip$_1$	432	20	40	38	6	0	32	10.29	760	634	114	0	520	250.41	9 120	6 262	620	0	5 642	11 469.83
Minepump	128	11	22	20	14	0	6	0.13	220	177	88	4	85	2.10	1 320	914	324	48	542	42.75
PKJab	72	12	24	20	12	0	8	0.08	264	177	60	2	115	1.79	1 760	919	166	18	735	39.66
Prevaylar	24	7	14	12	10	0	2	0.02	84	60	38	2	20	0.04	280	162	68	12	82	0.52
VCL	2 097 152	21	42	42	42	0	0	0.03	840	840	840	0	0	0.12	10 640	10 640	10 640	0	0	1.80
ZipMe	64	9	18	15	12	0	3	0.02	144	99	60	0	39	0.09	672	377	160	0	217	1.51
h264	1 152	17	34	30	14	0	16	50.95	544	416	96	0	320	1 567.02	5 440	3 542	448	0	3 094	49 977.64

For each subject system, we list the number of valid configurations ($\lvert\mathcal{V}\rvert$) and features ($\lvert\mathcal{F}\rvert$). Considering three sizes of partial configurations (PCs) ∂: $\lvert\text{supp}(\partial)\rvert = 1$ for 1-wise PC, $\lvert\text{supp}(\partial)\rvert = 2$ for 2-wise PC, and $\lvert\text{supp}(\partial)\rvert = 3$ for 3-wise PC), the second part of the table shows the number of PCs of that size ($\lvert\mathcal{P}\rvert$), with at least one valid configuration ($\lvert\mathcal{P}_\mathcal{V}\rvert$), that are directly observable ($\lvert\mathcal{D}_O\rvert$), that are indirectly observable ($\lvert\mathcal{I}_O\rvert$), that are non-observable ($\lvert\neg O\rvert$), and the overall time in seconds needed to compute the observability of the PCs.

of the systems, there are no indirectly observable partial configurations for the other half. For illustration, we show the share of partial configurations that are directly observable, indirectly observable, non-observable, and the share of partial configurations that have no valid representative in Figure 3. There is a clear trend that the share of directly observable partial configurations decreases with size for almost all systems. In contrast, the share of indirectly observable partial configurations increases with size for some systems, and the share of partial configurations that have no valid representative increases with the size for almost all systems. This is due to the fact that, for constrained configuration spaces, it is more likely that a partial configuration of a higher size has no valid representative.

Summarizing our findings for **RQ$_2$**, we conclude that the majority of partial configurations of sizes two and three are not observable, and, even for partial configurations of size one, there is a considerable share of partial configurations that are not observable.

5.3 Observability of Features Interactions

Interpretable blackbox performance models rely on partial configurations to explain the behavior of a configurable software system (see Section 2). In order to serve as an explanation (i.e., to be interpretable), the features and feature interactions appearing in a blackbox model must be observable; otherwise, the model cannot make reliable statements about their effects. In **RQ$_3$**, we verify the observability of features and feature interactions in state-of-the-art blackbox performance models of 11 subject systems from the literature. We extracted all features and feature interactions from these models (i.e., the individual terms in the model), and computed the observability of these partial configurations. Our results are summarized in Table 2. We see that the number of partial configurations used in the blackbox performance models ($\lvert\mathcal{P}\rvert$) ranges from only a few partial configurations (PostgreSQL includes only three) to a few hundred partial configurations (Lrzip$_2$ includes 220). Regarding size, we see that the majority of the models contain partial configurations of sizes up to three. The only exception is Opus, which includes partial configurations of size up to five. An interesting fact is that, while the number of features in the subject systems ($\lvert\mathcal{F}\rvert$) ranges from 14 to 60, the number of valid configurations ($\lvert\mathcal{V}\rvert$) ranges only between 180 and 6 480. That is, the valid configuration spaces are highly constrained. In terms of time for computing the observability of the partial configurations contained in the blackbox models (Time [s] column), it ranges from less than a second to a few hours, except for Opus, where the computation time is significantly higher. This is due to the fact that Opus includes partial configurations of size up to five, which consequently

Table 2: Statistics of general observability in state-of-the-art performance-influence models

System	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{P}_\mathcal{V}\rvert$	Max Size	$\lvert\mathcal{D}_O\rvert$	$\lvert\mathcal{I}_O\rvert$	$\lvert\neg O\rvert$	Time [s]
Brotli	180	30	166	166	2	0	0	166	21.85
FastDownward	347	60	41	41	3	0	0	41	144.22
HSQLDB	864	29	21	21	3	0	0	21	201.26
Lrzip$_2$	1 440	27	220	220	3	0	0	220	6 777.10
MariaDB	972	21	35	35	3	4	0	31	275.07
MySQL	972	21	25	25	3	4	0	21	144.08
OpenVPN	512	24	13	13	2	1	0	12	13.64
Opus	6 480	31	66	66	5	0	0	66	309 561.33
PostgreSQL	864	18	3	3	1	2	0	1	0.01
VP8	2 736	27	40	40	3	0	0	40	30 887.61
Z3	1 024	14	18	18	3	2	0	16	40.78

For each subject system, we list the number of valid configurations ($\lvert\mathcal{V}\rvert$), features ($\lvert\mathcal{F}\rvert$), partial configurations (PCs) ($\lvert\mathcal{P}\rvert$), PCs with at least one valid configuration ($\lvert\mathcal{P}_\mathcal{V}\rvert$), and the maximum size of PCs. The right part of the table lists the number of PCs that are directly observable ($\lvert\mathcal{D}_O\rvert$), indirectly observable ($\lvert\mathcal{I}_O\rvert$), non-observable ($\lvert\neg O\rvert$), and the total time in seconds needed to compute the observability of the PCs.

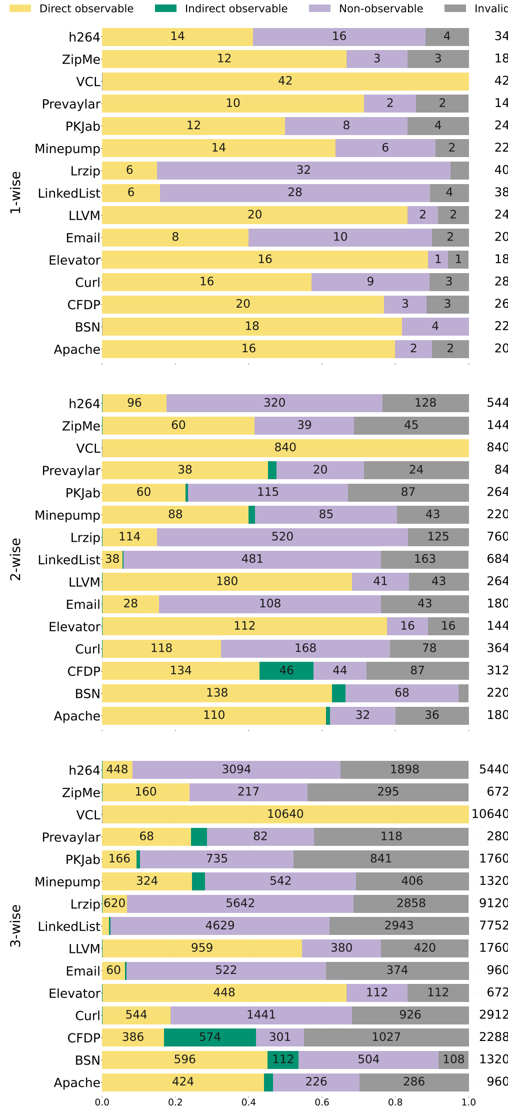

Figure 3: For each subject system and size of partial configuration (PC), we show in each bar the share of PCs that are directly observable, indirectly observable, non-observable, and the share of PCs that have no valid representative configuration. The absolute numbers of PCs of each type are included in the corresponding bar, and the total number of PCs of the corresponding size and system is shown right next of the bar.

leads to a significantly higher number of partitions that have to be checked.

While all models include only partial configurations that have at least one valid representative ($|\mathcal{P}_{\mathcal{V}}|$), the number of directly observable partial configurations ($|\mathcal{D}_{O}|$) is significantly lower, even zero for half of the models. The number of indirectly observable[9] partial configurations ($|\mathcal{I}_{O}|$) is zero in all models. That is, the vast majority of the partial configurations used in our subject blackbox models are non-observable ($|\neg O|$), which is a surprising result that we discuss in Section 6.

Summarizing our findings for **RQ**$_3$, we conclude that the majority of features and feature interactions used in state-of-the-art blackbox performance models are not observable.

6 DISCUSSION

In this section, we discuss the implications of our results, potential threats to validity, and related work.

6.1 Research Questions

Deciding which partial configurations of a configurable software system are blackbox observable is a combinatorial problem, since the number of partial configurations grows exponentially with the number of features, in the worst case. That is, the time needed to compute the observability of *all* partial configurations of a configurable software system may be easily infeasbile in practice. However, as our first major result shows, even for non-trivial feature models, the observability of partial configurations of sizes up to three covering all pair-wise and triple-wise feature interactions, can be computed in reasonable time. Since most test analysis methods stay within these bounds [1], we consider the decision procedure for observability effectively computable, despite the worst-case computational complexity (see Section 3).

A second major result is that a large fraction of partial configurations is non-observable for the majority of our subject systems. This means a large fraction of partial configurations cannot be used to explain the behavior of a given system. Given the highly constrained nature of contemporary configurable software systems [43], this is not surprising. What is concerning is that a growing number of reasoning and analysis methods rely on partial configurations to *explain* system behavior (e.g., the presence of a feature interaction). As an example, blackbox performance models [30] have been used to explain the performance behavior of a given configurable software system by interpreting the coefficients of individual model terms (cf. Section 2) as performance influences of the corresponding partial configurations (i.e., single features or feature interactions) [34, 40, 41, 52, 60], which leads to the next major result.

A third major result is that a substantial fraction of model terms (i.e., corresponding partial configurations) of state-of-the-art blackbox performance models is non-observable. This begs the question of how reliable explanations based on influences of individual features and feature interactions actually are. Consider the following example, for illustration.

Example 6.1. Let us assume a configurable software system with three features e, r, and s, and two configurations $c_1 = \{e \mapsto \top, r \mapsto \top, s \mapsto \top\}$ and $c_2 = \{e \mapsto \top, r \mapsto \top, s \mapsto \bot\}$. The observed performance of c_1 is 10s and of c_2 is 5s. A performance model with a minimal prediction error that is consistent with these two observations is $\Pi = 5 \cdot e + 5 \cdot s$. Note, however, that only the partial configuration $\partial_s = \{s \mapsto \top\}$ is observable, whereas $\partial_e = \{e \mapsto \top\}$ is not. That is, the influence of feature e on the system's performance cannot be *reliably* inferred from Π.

[9] Recall, to improve the interpretability of the numbers, we report the number of generally observable partial configurations that are *not* directly observable.

To put our results in perspective, our experiments suggest that non-observable features and feature interactions are prevalent in practice and must be taken into account when interpreting blackbox models and analysis results in this field. Our notion of observability and effective decision procedure raises the question whether current blackbox analysis methods and models can reliably explain configurable software systems behavior. This may provide the basis for the development of more interpretable blackbox analysis methods and models for configurable software systems.

Influence of Constraints on Observability. One important point to highlight is that it is expected for configurable software systems to have a small share of observable partial configurations. The share of observable partial configurations tends to decrease with the number of constraints in the feature model. This is similar to the share of valid configurations, which also decreases with the number of constraints. One common property of all configurable software systems is that the share of valid configurations in the configuration space is small. Therefore, we expect the share of observable partial configurations in the space of all partial configurations to be small, as well.

To illustrate this, consider our subject system PKJab as a representative example (see Table 1). Even though it has only 12 features, the share of valid configurations is small and consists of less than 2% of the total of possible configurations. For illustration, we compute the share of observable partial configurations of size up to three in the PKJab feature model, and compare it to the share of valid configurations. To understand the influence of constraints on observability, we compute the share of observable partial configurations in variations of the PKJab feature model, where some (or all) constraints are removed. We start from a version of the PKJab feature model with no constraints, and add each constraint in a stepwise manner, until we arrive at the actual feature model that was used in our experiments. Figure 4 shows the results of computing observability for each step. Not surprisingly, everything is observable and all configurations are valid on the initial model (0 constraints). As we progress in adding constraints, such as mandatory features and implications[10], the share of valid configurations in the configuration space decreases. The share of partial configurations that are observable also decreases, and we clearly see the distinction in the observability ratio of 1-wise, 2-wise, and 3-wise partial configurations. While Figure 4 illustrates the results for PKJab only, similar graphs can be obtained for other highly constrained systems from our evaluation. In other words, though it might appear counterintuitive that most partial configurations are non-observable, it is to be expected that configurable software systems only have a small share of observable partial configurations.

Scalability. Deciding whether a partial configuration ∂ is observable according to Algorithm 1 or Algorithm 2 is inherently expensive: the algorithm to check for direct observability (Algorithm 1) iterates over all valid configurations covered by ∂, which could be exponentially many. Even worse, the algorithm for general observability (Algorithm 2) iterates over all possible partitions of the support of ∂, each time checking direct observability several

[10]First, we add all mandatory features as constraints, then all implications in their simplest form. While other orders are possible, the general result would not change.

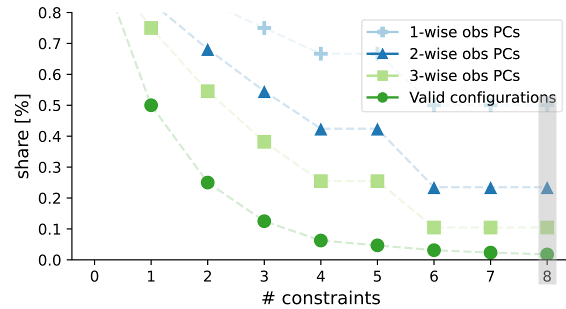

Figure 4: The share of partial configurations (PCs) of a given size (color) that are observable (y-axis) under the given constraints. To compare, we plot the fraction of configurations that are valid (i.e., satisfy all constraints). The x-axis shows the number of added constraints to the configuration space, starting with no constraints. The actual results of the system are highlighted with a gray box (#constraints = 8).

times. To render our implementations effective, we exploited binary decision diagrams (BDDs, [5]) as concise data structure to represent partial configurations. While BDDs are usually compact and allow for efficient manipulation, their size heavily depends on the specific structure of partial configurations. In our experiments, we focused on answering the research questions concerning all possible partial configurations, which required checking observability exhaustively. We would like to highlight that in practice, we imagine such exhaustive checks usually not being performed, e.g., when to determine whether it is worth to run tests on a specific partial configuration where checking observability beforehand could reduce the test space. Especially direct observability checks performed almost instantly in our experiments, witnessing the instance-based scalability of our approach. Dedicated algorithms for exploiting structural BDD properties for counterfactual reasoning BDDs [17, 27] or parallelizing BDD operations [14, 29, 57] could further improve scalability also for checking general observability.

6.2 Threats to Validity

Internal Validity. A threat to internal validity arises from the fact that our decision procedure is based on the constraints provided by the feature models of our subject systems. Therefore, the correctness of our observability results relies on the correctness of the feature models. To mitigate this threat, we have used a diverse set of feature models from the literature and real-world systems that have been used in previous work. A further threat arises from the selection of our subject systems. While we cannot rule out specific characteristics of our subject systems that might bias our results, our diverse set of subject systems and community benchmarks from the literature mitigates this threat.

External Validity. While it is difficult to reliably state to what extent our results are valid for configurable software systems beyond the ones we have considered, we are confident that our results are representative and generalizable to a sufficient extent for scholarly discussion. To attain a reasonable level of external validity, we have

selected a diverse set of subject systems from different areas, including real-world systems from a variety of domains, as well as community benchmarks from the literature. A further threat arises from the decision to use blackbox performance models as a basis for answering our third research question. Clearly, there are other kinds of blackbox models and other types of blackbox reasoning and analysis methods for feature interaction detection (see Section 6.3). While the models we use are representative for a whole research area, it is not our intention to make definitive statements about all kinds of models and methods. Rather, we provide evidence that the lack of observability *may* substantially affect the results of blackbox analysis methods and models.

6.3 Related Work

Various techniques have been proposed to analyze and reason about the behavior of configurable software systems. Many of these techniques are concerned with detecting performance regressions [35] and feature interactions [10, 17, 53] in the systems under investigation to model their behavior, and explain their effects.

Feature Modeling. Batory was among the first to represent feature models in propositional logic [4]. Schobbens et al. [50] survey different notations for feature models and establish a formal, concise, and generic definition of feature models that encompasses feature-oriented domain analysis, proposed by Kang et al. [33]. The goal of Schobbens et al. is in line with the goal of Classen et al. [9], who clarify the notion of a feature and feature interactions, allowing early identification of feature interactions in the systems' environment. While these methods ensure interpretable models, they are costly, time-consuming, and rely on the knowledge of domain experts.

Performance Modeling of Configurable Software Systems. A variety of techniques have been proposed to model the performance of a configurable software system based on the performance influences of its features. Blackbox analysis techniques are able to study the behavior of a system based only on observations. To model the performance of a system, a variety of blackbox performance modeling techniques have been proposed. These techniques range from linear regression [32, 52, 53], regression tree-based methods [22, 23, 48], Fourier learning [25, 66], and deep neural networks [8, 24] to probabilistic programming [12]. A related line of work is concerned with finding (near) optimal configurations regarding performance [7, 28, 44, 49]. All these methods assume that the performance of a system can be modeled by blackbox observations only. However, to the best of our knowledge, none of these methods consider the observability of features and feature interactions in their models; instead they rely on statistical methods to find the best model in terms of prediction accuracy. In contrast, whitebox analysis techniques examine the behavior of a system based on its source code and can explain the behavior of features and feature interactions [58, 59, 63]. However, whitebox analysis techniques often do not scale in practice, suffer from a significant number of false positives, or require sophisticated setups and tools.

Feature Interaction Detection. Over the years, several studies have analyzed the state of the art of feature interactions in software engineering and identify open research questions [1, 6, 36, 45, 54]. Apel et al. [3] explore the nature of feature interactions and their internal and external manifestations. Our notion of observability applied to feature interactions aligns with this classification. Siegmund et al. [52, 53] propose different heuristics to identify performance-relevant feature interactions using blackbox measurements. A line of research has followed in their footsteps [8, 12, 22–25, 32, 34, 40–42, 48, 60, 66]. Some of these approaches use blackbox performance models to explain the behavior of a system [34, 40, 41, 60]. Whitebox analysis techniques can also identify feature interactions [58, 59, 63, 64], but are outside the scope of this paper. Shaker [51] presents a similar approach, defining a feature interaction taxonomy that uses formally modeled product-line requirements in a world model to detect feature interactions. Fantechi et al. [19] propose a definition framework for functional feature interactions within which they show that a 3 (or greater)-way functional feature interaction is always caused by a 2-way functional feature interaction. Similarly, Fischer et al. [20] propose a heuristics to reduce the search space for feature interactions by focusing only on those that might be caused by lower order feature interactions. Interestingly, Garvin and Cohen [21] present a formal definition of feature interaction faults that combines the advantages of blackbox and whitebox analyses. They propose guiding whitebox analysis to detect feature interaction faults based on blackbox analysis results. Dubslaff et al. [17] propose the notion of feature causes to identify the causes of a predefined, emergent behavior (effect) of a configurable software system. Thereupon, they established a relation of feature causes to feature interactions. Our notion of observability is not limited to feature interaction detection, but can be used together with existing methods to improve or verify their results (see Section 6.1).

7 CONCLUSION

The research community has proposed several blackbox approaches for detecting feature interactions and inferring effects of individual features and their interactions on system behavior. The crux is that, without any information on the inner workings of the system in question, it is rather difficult to pinpoint actual effects and influences of individual features and feature interactions. Worse, certain effects may not be observable at all, meaning they cannot be inferred by running and comparing observations from different configurations of the system. Combined with our empirical results, this raises the question of to what extent blackbox analysis methods and models actually produce interpretable results.

In this paper, we devise a *decision procedure* to verify whether the effect of a given feature or potential feature interaction can be isolated by *blackbox observations* of a set of system configurations. We introduce the notion of *general observability*, inspired by *counterfactual reasoning* about configuration decisions [17]. Based on the given constraints of the configuration space, we reason and decide whether it is possible to observe the effect of a set of features in a blackbox fashion. *Direct* observability requires a single configuration as a reference for observing the effect, whereas *general* observability relaxes this requirement and suffices with a set of configurations as a reference that jointly cover all features in the set. In a series of experiments on community benchmarks and real-world systems, we found that (1) general observability is indeed feasible in real-world settings, (2) constraints in real-world configuration

spaces often limit observability, and (3) blackbox performance models often include effects that are *de facto* not observable.

Putting our results in perspective, our notion of observability and our effective decision procedure lay the foundation for developing more interpretable blackbox analysis methods and models for configurable software systems. Our empirical results suggest that non-observable features and feature interactions are prevalent in practice and need to be considered when interpreting blackbox analysis results in this field.

ACKNOWLEDGMENTS

The authors are supported by the DFG through the Collaborative Research Center TRR 248, project ID 389792660 (https://perspicuous-computing.science) and Cluster of Excellence CeTI EXC 2050/1, project ID 390696704 (https://ceti.one), the NWO through Veni grant VI.Veni.222.431, as well as CNPq (grants 315532/2021-1 and 423125/2021-4), CAPES (88881.512952/2020-01), Alexander von Humboldt Foundation, and INES[11] (CNPq grant 465614/2014-0, CAPES grant 88887.136410/2017-00, and FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03/14).

REFERENCES

[1] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. 2014. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Reports* 4, 7 (2014), 1–24. https://doi.org/10.4230/DagRep.4.7.1

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-oriented software product lines*. Springer.

[3] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *5th International Workshop on Feature-Oriented Software Development, FOSD '13, Indianapolis, IN, USA, October 26, 2013*, Andreas Classen and Norbert Siegmund (Eds.). ACM, 1–8. https://doi.org/10.1145/2528265.2528267

[4] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3714)*, J. Henk Obbink and Klaus Pohl (Eds.). Springer, 7–20. https://doi.org/10.1007/11554844_3

[5] Randal E. Bryant. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *Comput. Surveys* 24, 3 (Sept. 1992), 293–318. https://doi.org/10.1145/136035.136043

[6] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141. https://doi.org/10.1016/S1389-1286(02)00352-3

[7] Tao Chen and Miqing Li. 2021. Multi-objectivizing software configuration tuning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 453–465.

[8] Jiezhu Cheng, Cuiyun Gao, and Zibin Zheng. 2023. HINNPerf: Hierarchical interaction neural network for performance prediction of configurable systems. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–30.

[9] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. 2008. What's in a Feature: A Requirements Engineering Perspective. In *Fundamental Approaches to Software Engineering*, José Luiz Fiadeiro and Paola Inverardi (Eds.). Springer Berlin Heidelberg, 16–30.

[10] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, and C.J. Colbourn. 2003. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering, 2003. Proceedings*. 38–48. https://doi.org/10.1109/ICSE.2003.1201186

[11] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2013. ProVeLines: a product line of verifiers for software product lines. In *Proceedings of the 17th Systems and Software Product Line Conference (SPLC)*. ACM, 141–146.

[12] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2020. Mastering uncertainty in performance estimations of configurable software systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 684–696.

[13] Clemens Dubslaff, Kai Ding, Andrey Morozov, Christel Baier, and Klaus Janschek. 2019. Breaking the Limits of Redundancy Systems Analysis. In *Proceedings of the 29th European Safety and Reliability Conference (ESREL)*. 2317–2325.

[11] https://www.ines.org.br

[14] Clemens Dubslaff, Nils Husung, and Nikolai Käfer. 2024. Configuring BDD Compilation Techniques for Feature Models. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference - Volume A* (Dommeldange, Luxembourg) *(SPLC '24)*. Association for Computing Machinery, New York, NY, USA, 209–216. https://doi.org/10.1145/3646548.3676538

[15] Clemens Dubslaff and Maximilian A Köhl. 2022. Configurable-by-construction runtime monitoring. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 220–241.

[16] Clemens Dubslaff, Andrey Morozov, Christel Baier, and Klaus Janschek. 2020. Reduction Methods on Error-Propagation Graphs for Quantitative Systems Reliability Analysis. In *Proceedings of the 30th European Safety and Reliability Conference (ESREL) and 15th Probabilistic Safety Assessment and Management Conference (PSAM)*.

[17] Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel. 2024. Feature causality. *Journal of Systems and Software* 209 (2024), 111915. https://doi.org/10.1016/j.jss.2023.111915

[18] C. J. Ducasse. 1926. On the Nature and the Observability of the Causal Relation. *The Journal of Philosophy* 23, 3 (1926), 57–68. http://www.jstor.org/stable/2014377

[19] Fantechi, Alessandro and Gnesi, Stefania and Semini Laura. 2017. Optimizing Feature Interaction Detection. In *Critical Systems: Formal Methods and Automated Verification*, Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti (Eds.). Springer International Publishing, Cham, 201–216.

[20] Stefan Fischer, Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2018. Predicting Higher Order Structural Feature Interactions in Variable Systems. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 252–263. https://doi.org/10.1109/ICSME.2018.00035

[21] Brady J. Garvin and Myra B. Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE)*. ACM, 90–99.

[22] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[23] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empirical Software Engineering* 23 (2018), 1826–1867.

[24] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance prediction for configurable software with deep sparse neural network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1095–1106.

[25] Huong Ha and Hongyu Zhang. 2019. Performance-influence model for highly configurable software with fourier learning and lasso regression. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 470–480.

[26] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2017. Test them all, is it worth it? A ground truth comparison of configuration sampling strategies. *arXiv preprint arXiv:1710.07980*, 1–8.

[27] Hans Harder, Simon Jantsch, Christel Baier, and Clemens Dubslaff. 2023. A Unifying Formal Approach to Importance Values in Boolean Functions. In *IJCAI*. ijcai.org, 2728–2737. https://doi.org/10.24963/ijcai.2023/304

[28] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 517–528. https://doi.org/10.1109/ICSE.2015.69

[29] Nils Husung, Clemens Dubslaff, Holger Hermanns, and Maximilian A. Köhl. 2024. OxiDD: A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust. In *TACAS*. Springer. https://doi.org/10.1007/978-3-031-57256-2_13

[30] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The Interplay of Sampling and Machine Learning for Software Performance Prediction. *IEEE Softw.* 37, 4 (2020), 58–66. https://doi.org/10.1109/MS.2020.2987024

[31] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 1084–1094.

[32] Christian Kaltenecker, Stefan Mühlbauer, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2023. Performance Evolution of Configurable Software Systems: An Empirical Study. *Empirical Software Engineering (EMSE)* (2023). To appear.

[33] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[34] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in modeling performance of highly configurable software systems. *Softw. Syst. Model.* 18, 3 (2019), 2265–2283. https://doi.org/10.1007/s10270-018-0662-9

[35] Donghun Lee, Sang K Cha, and Arthur H Lee. 2011. A performance anomaly detection and analysis framework for dbms development. *IEEE Transactions on Knowledge and Data Engineering* 24, 8 (2011), 1345–1360.

[36] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2023. Input sensitivity on the performance of configurable systems an empirical study. *Journal of Systems and Software* 201 (2023), 111671.

[37] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebraic Methods Program.* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[38] David Lewis. 1981. *Counterfactuals and Comparative Possibility.* Springer Netherlands, Dordrecht, 57–85. https://doi.org/10.1007/978-94-009-9117-0_3

[39] Yang-Yu Liu, Jean-Jacques Slotine, and Albert-László Barabási. 2013. Observability of complex systems. *Proceedings of the National Academy of Sciences* 110, 7 (2013), 2460–2465. https://doi.org/10.1073/pnas.1215508110 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1215508110

[40] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2020. Identifying Software Performance Changes Across Variants and Versions. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* IEEE, 611–622. https://doi.org/10.1145/3324884.3416573

[41] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 2085–2097. https://doi.org/10.1109/ICSE48619.2023.00176

[42] Daniel-Jesus Munoz, Mónica Pinto, and Lidia Fuentes. 2023. Detecting feature influences to quality attributes in large and partially measured spaces using smart sampling and dynamic learning. *Knowledge-Based Systems* 270 (2023), 110558.

[43] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. Software Eng.* 41, 8 (2015), 820–841. https://doi.org/10.1109/TSE.2015.2415793

[44] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Trans. Software Eng.* 46, 7 (2020), 794–811. https://doi.org/10.1109/TSE.2018.2870895

[45] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. 2008. Feature interaction: The security threat from within software systems. *Progress in Informatics* 5, 75 (2008), 1.

[46] C. R. Ramakrishnan and R. Sekar. 2002. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security* 10, 1-2 (2002), 189–209. https://doi.org/10.3233/JCS-2002-101-209

[47] Genaína Nunes Rodrigues, Vander Alves, Vinicius Nunes, André Lanna, Maxime Cordy, Pierre-Yves Schobbens, Amir Molzam Sharifloo, and Axel Legay. 2015. Modeling and Verification for Probabilistic Properties in Software Product Lines. In *Proceedings of the 16th Symposium on High Assurance Systems Engineering (HASE).* IEEE, 173–180.

[48] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 342–352.

[49] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany H. Ammar. 2013. Scalable product line configuration: A straw to break the camel's back. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 465–474. https://doi.org/10.1109/ASE.2013.6693104

[50] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Computer Networks* 51, 2 (2007), 456–479. https://doi.org/10.1016/j.comnet.2006.08.008 Feature Interaction.

[51] Pourya Shaker. 2013. A feature-oriented modelling language and a feature-interaction taxonomy for product-line requirements. (2013).

[52] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* ACM, 284–294.

[53] Norbert Siegmund, Sergiy Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).* 167–177. https://doi.org/10.1109/ICSE.2012.6227196

[54] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2018. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology* 98 (2018), 44–58. https://doi.org/10.1016/j.infsof.2018.01.016

[55] Fabio Somenzi. 1998. CUDD: CU decision diagram package release 2.3. 0. *University of Colorado at Boulder* 621 (1998).

[56] Eduardo D. Sontag. 1984. A concept of local observability. *Systems & Control Letters* 5, 1 (1984), 41–47. https://doi.org/10.1016/0167-6911(84)90007-0

[57] Tom van Dijk and Jaco van de Pol. 2017. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer* 19, 6 (2017), 675–696. https://doi.org/10.1007/s10009-016-0433-2

[58] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. ConfigCrusher: towards white-box performance analysis for configurable systems. *ASE* 27 (2020), 265–300. https://doi.org/10.1007/s10515-020-00273-8

[59] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 1072–1084. https://doi.org/10.1109/ICSE43902.2021.00100

[60] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2022. On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1571–1583. https://doi.org/10.1145/3510003.3510043

[61] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proceedings of the 37th International Conference on Software Engineering (ICSE).* IEEE, 178–188.

[62] Sandra Wachter, Brent D. Mittelstadt, and Chris Russell. 2017. Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR. *Harvard Journal of Law and Technology* 31 (2017), 841–887.

[63] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-Box Performance-Influence Models: A Profiling and Learning Approach (Replication Package). In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021.* IEEE, 232–233. https://doi.org/10.1109/ICSE-Companion52605.2021.00107

[64] Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster variational execution with transparent bytecode transformation. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 117:1–117:30. https://doi.org/10.1145/3276487

[65] Franz Wotawa, Gerhard Friedrich, and Artur Andrzejak. 2018. Software configuration diagnosis? A Survey of existing methods and open challenges. http://confws.ist.tugraz.at 20th International Workshop on Configuration, ConfWS ; Conference date: 27-09-2018 Through 28-09-2018.

[66] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance prediction of configurable software systems by fourier learning (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 365–373.