Causality in Configurable Software Systems

Clemens Dubslaff
clemens.dubslaff@tu-dresden.de
Centre for Tactile Internet with Human-in-the-Loop (CeTI)
Technische Universität Dresden
Dresden, Germany

Kallistos Weis
kallistos@cs.uni-saarland.de
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany

Christel Baier
christel.baier@tu-dresden.de
Technische Universität Dresden
Dresden, Germany

Sven Apel
apel@cs.uni-saarland.de
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany

ABSTRACT

Detecting and understanding reasons for defects and inadvertent behavior in software is challenging due to their increasing complexity. In configurable software systems, the combinatorics that arises from the multitude of features a user might select from adds a further layer of complexity. We introduce the notion of *feature causality*, which is based on *counterfactual reasoning* and inspired by the seminal definition of actual causality by Halpern and Pearl. Feature causality operates at the level of system configurations and is capable of identifying features and their interactions that are the reason for emerging functional and non-functional properties. We present various methods to explicate these reasons, in particular well-established notions of *responsibility* and *blame* that we extend to the feature-oriented setting. Establishing a close connection of feature causality to prime implicants, we provide algorithms to effectively compute feature causes and causal explications. By means of an evaluation on a wide range of configurable software systems, including community benchmarks and real-world systems, we demonstrate the feasibility of our approach: We illustrate how our notion of causality facilitates to identify root causes, estimate the effects of features, and detect feature interactions.

1 INTRODUCTION

Configurable software systems offer a wide variety of configuration options that control the activation of features desired by the user and that influence critical functional and non-functional properties such as performance. The often huge configuration spaces render

the detection, prediction, and explanation of defects and inadvertent behavior challenging tasks. While there are specifically tailored analysis methods to tackle this challenge [85], research on their *explainability* is still in its infancy [8]. The potentially exponential number of system configurations and corresponding analysis results, bug reports, or other feature-dependent properties demand techniques for a meaningful and feasible interpretation.

In this paper, we present a set of fundamental concepts and methods to identify and interpret properties of configurable systems *at the level of features* by *causal reasoning*. We introduce the notion of *feature causes* as feature selections that are the reason for emergent system behaviors. Our notion of feature causality is inspired by the seminal counterfactual definition of *actual causality* by Halpern and Pearl [45, 46]. Relevant analysis and reasoning tasks that we address include to determine features that cause a bug, the degree to which some configurations are responsible for bad system performance, or which features necessarily have to interact for inadvertent behavior.

Since features correspond to system functionalities specified by software engineers, they often have a dedicated meaning in the target application domain [4]. To this end, defects (and other behaviors of interest) detected at the level of features can provide important insights for the resolution of *variability bugs* [1, 40, 72] and *configuration-dependent behavior* [44, 63, 77, 78]. As such, they are certainly more informative and actionable than low-level program traces alone. Developers may choose to focus on those feature implementations identified as root causes of bugs or simply disallow or coordinate the activation of certain features when defects are related to them.

Our presented techniques are generic in the sense that they are neither language-, architecture-, nor environment-specific and applicable within any effective method to analyze or test variability-aware properties. To this end, causal reasoning on both variability-aware white-box and black-box analyses is supported. This complements existing causal reasoning techniques for the detection of root causes: Approaches such as delta-debugging [24, 95], causal testing [51], or causal trace analysis [11] require a white-box analysis that operates at the level of code and are not variability-aware. Hence, they usually would have to be applied on a multitude of system configurations for a variability-aware causal analysis, suffering from a combinatorial blowup. We envision applications of feature causality at those development phases where analysis methods are

Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel

used, for instance, in software product line engineering. Also in production-level deployments our techniques shall be useful to optimize software through causally relevant configurations.

Evaluation. We present algorithms to compute feature causes, represent, and interpret them by means of concise logic formulas, feature interactions, and *responsibility* and *blame* [17]. Our prototypical implementation relies on *binary decision diagrams (BDDs)* [13] and the computation of prime implicants using the de-facto standard two-level logic minimizer ESPRESSO [61]. By means of an analysis of several configurable systems, including community benchmarks and real-world systems, we investigate feature causes and their properties. We demonstrate that our notion of feature causes and methods to represent them help to pinpoint features relevant for the configurable system's properties and illustrate how feature interactions can be detected and quantified.

Contributions. In summary, our contributions are:

- We introduce the notion of feature causality inspired by the well-established counterfactual definition of actual causality by Halpern and Pearl [45, 46].
- We show that feature causes coincide with certain prime implicants, leading to an algorithm to compute all feature causes.
- We provide methods to interpret and represent feature causes by propositional formulas, responsibility and blame, and potential feature interactions.
- We offer a BDD-based prototype to compute and represent feature causes and feature interactions.
- We conduct several experiments illustrating how to determine and reason about feature causes in different realistic settings.

Supplement. Proofs of the theoretical statements are included in [30]. The source code of our implementation and raw data to reproduce our experiments are publicly available [36].

2 BACKGROUND

In this section, we revisit basic concepts and notions from logics and configurable systems used throughout the paper.

Interpretations. A *partial interpretation* over a set X is a partial mapping $\partial \colon X \rightharpoonup \{\texttt{true}, \texttt{false}\}$. We denote by $\operatorname{supp}(\partial)$ the *support* of ∂, i.e., the set of all elements $x \in X$ where $\partial(x)$ is defined. We say that ∂ is a *total interpretation* if $\operatorname{supp}(\partial) = X$ and denote by $\Delta(X)$ and $\Theta(X)$ the set of partial and total interpretations, respectively. Given a partial interpretation $\partial \in \Delta(X)$, we define its semantics $[\![\partial]\!] \subseteq \Theta(X)$ as the set of all total interpretations $\theta \in \Theta(X)$ where for all $x \in \operatorname{supp}(\partial)$ we have $\partial(x) = \theta(x)$. We say that $\partial \in \Delta(X)$ *covers* $\partial' \in \Delta(X)$ if $[\![\partial']\!] \subseteq [\![\partial]\!]$. For a set of partial interpretations $\mathcal{P} \subseteq \Delta(X)$, we define $[\![\mathcal{P}]\!] = \bigcup_{\partial \in \mathcal{P}} [\![\partial]\!]$. The *$x$-expansion* of a partial interpretation $\partial \in \Delta(X)$ is the partial interpretation $\partial{\uparrow}_x \in \Delta(X)$ where $\operatorname{supp}(\partial{\uparrow}_x) = \operatorname{supp}(\partial) \backslash \{x\}$ and where $\partial{\uparrow}_x(y) = \partial(y)$ for all $y \in \operatorname{supp}(\partial) \backslash \{x\}$. For a set of total interpretations $\mathcal{T} \subseteq \Theta(X)$, an *implicant* of $\mathcal{T}$ is a partial interpretation $\partial \in \Delta(X)$ where $[\![\partial]\!] \subseteq \mathcal{T}$. We call ∂ a *prime implicant* if ∂ is minimal, i.e., $[\![\partial{\uparrow}_x]\!] \not\subseteq \mathcal{T}$ for all $x \in \operatorname{supp}(\partial)$.

Propositional logics. A *propositional logic formula* over a set X is an expression defined by the grammar

$$\phi \;=\; \texttt{true} \mid \texttt{false} \mid x \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$$
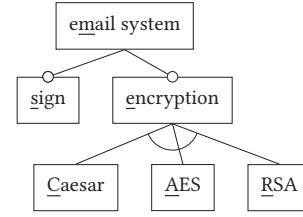

Figure 1: Feature diagram for the email system example

where x ranges over X. The *length* $|\phi|$ of a formula ϕ is recursively defined by $|\texttt{true}| = |\texttt{false}| = |x| = 1$, $|\neg\phi| = |\phi| + 1$, and $|\phi_0 \wedge \phi_1| = |\phi_0 \vee \phi_1| = |\phi_0| + |\phi_1| + 1$. For a partial interpretation $\partial \in \Delta(X)$, we write $\partial \models \phi$ if either $\phi = \texttt{true}$, $\phi = x \in \operatorname{supp}(\partial)$ and $\partial(x) = \texttt{true}$, $\phi = \neg\psi$ and not $\partial \models \psi$, $\phi = \phi_0 \wedge \phi_1$ and $\partial \models \phi_0$ and $\partial \models \phi_1$, and $\phi = \phi_0 \vee \phi_1$ and $\partial \models \phi_0$ or $\partial \models \phi_1$. The semantics of ϕ is the set of all satisfying total interpretations $[\![\phi]\!] = \{\theta \in \Theta(X) : \theta \models \phi\}$.

Configurable systems. A widely adopted concept to model configurable systems is by means of features [4]. *Features* encapsulate optional or incremental units of functionality [94] and are used to describe commonalities and variabilities of whole families of systems. At an abstract level, we fix a set F of Boolean configuration options where each element corresponds to some feature of the system. We call a total interpretation $\theta \in \Theta(F)$ over F a *configuration*, which we usually describe by listing the selected features, i.e., the features $x \in F$ where $\theta(x) = \texttt{true}$. The set of *Valid configurations* $\mathcal{V} \subseteq \Theta(F)$ comprises those configurations for which there exists a corresponding system implementation, usually specified by a *feature diagram* [54]. A partial interpretation over F is called *partial configuration.*

Example 2.1. As the running example, consider a simple email system over features $F = \{m, s, e, c, a, r\}$, formalizing the base email system functionality, optional features for signing and encryption, and encryption methods Caesar, AES, and RSA. For the encryption features, we assume that exactly one can be selected. The described variability constraints for the email system are specified in the feature diagram shown in Figure 1, leading to valid configurations

$$\mathcal{V} \;=\; \{\, m, mec, mea, mer, ms, msec, msea, mser \,\}.$$

3 FEATURE CAUSALITY

The notion of *causality* has been extensively studied in philosophy, social sciences, and artificial intelligence [37, 42, 66, 92]. We focus on *actual causality*, describing binary causal relationships between cause events C and effect events E. Halpern and Pearl formalized actual causality based on the concept of *counterfactual dependencies* [57] using a structural-equation approach [45–47]. The idea of counterfactual reasoning [90] relies on the assumption that E would not have happened if C had not happened before, which corresponds to the standard "but-for" test used in law.

In this section, we take inspiration of the definition by Halpern and Pearl [45, 46] to establish a notion of causality at the level of features. Here, we interpret the selection of features as events considered for actual causality. The basic reasoning task we address then amounts to *determine those feature selections that cause a given*
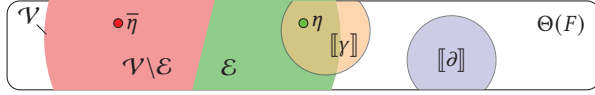
Figure 2: Configuration sets for feature causality

effect property. Examples for effect properties are "the execution time is longer than five minutes" or "the system crashes".

We assume to have described the effect properties as *effect set* $\mathcal{E} \subseteq \mathcal{V}$ of valid configurations for which an effect property can be observed. Elements of $\mathcal{E}$ are called *effect instances*. All other valid configurations in $\mathcal{V} \backslash \mathcal{E}$ are assumed not to exhibit the effect. Feature selections are naturally specified by partial configurations. Clearly, a partial configuration γ can only be a cause of the effect if γ ensures the effect to emerge, i.e., all valid configurations covered by γ are effect instances. Furthermore, following counterfactual reasoning, we require for γ being a cause that, if we would select features of γ differently, there might be a configuration for which the effect does not emerge. These two intuitive conditions on causality are reflected in our formal definition of causes of $\mathcal{E}$ w.r.t. $\mathcal{V}$:

Definition 3.1. A *feature cause* of an effect $\mathcal{E}$ w.r.t. valid configurations $\mathcal{V}$ is a partial configuration $\gamma \in \Delta(F)$ where

(FC$_1$) $\varnothing \neq [\![\gamma]\!] \cap \mathcal{V} \subseteq \mathcal{E}$ and
(FC$_2$) $[\![\gamma\!\uparrow_x]\!] \cap \mathcal{V} \not\subseteq \mathcal{E}$ for all $x \in \mathrm{supp}(\gamma)$.

We denote by $\mathrm{Causes}(\mathcal{E}, \mathcal{V})$ the set of all causes for $\mathcal{E}$ w.r.t. $\mathcal{V}$.

In case **FC$_1$** holds for a partial configuration γ, we say that γ is *sufficient* for $\mathcal{E}$ w.r.t. $\mathcal{V}$ [40]. The counterfactual nature of **FC$_2$** ensures that for every feature cause γ and $x \in \mathrm{supp}(\gamma)$ there is a *counterfactual witness* $\bar{\eta} \in [\![\gamma\!\uparrow_x]\!] \cap (\mathcal{V} \backslash \mathcal{E})$. That is, a valid feature configuration where the effect does not emerge but where changing one feature selection may yield an effect instance. Note that **FC$_2$** ensures minimality of the feature cause w.r.t. its support, i.e., dropping conditions on interpretations of features necessarily leads to a partial configuration that is not sufficient for the effect anymore. In the formal definition of Halpern and Pearl causality [45], counterfactuality and minimality are stated in two distinct conditions.

We usually denote configurations in $\mathcal{E}$ by η, counterfactual witnesses in $\mathcal{V} \backslash \mathcal{E}$ by $\bar{\eta}$, and feature causes by γ. Figure 2 depicts the relation between valid configurations, effects, causes, and counterfactual witnesses.

Example 3.2. Let us continue our running example of the configurable email system introduced in Example 2.1. We consider an effect property reflecting "long decipher time", e.g., that it takes in average more than three months for an attacker to decrypt an email. Assume that this effect property can be observed by configurations in which AES or RSA are selected, i.e., $\mathcal{E} = \{mea, mer, msea, mser\}$. Conversely, in all valid configurations in which AES and RSA are not selected, the effect does not emerge. In this setting, the encryption features AES and RSA are both causes since all valid configurations with either feature show the effect. Considered in isolation, AES and RSA are not necessary for the effect, as one can choose the other encryption feature (RSA or AES, respectively) to ensure the effect. The sign feature does not trigger the effect and is not a cause.

Interestingly, a further cause is given by selecting the encryption feature and explicitly deselecting the Caesar feature, illustrating

that also explicitly not selecting features might be a cause of some effect. This hints at the fact that causes can be represented in different ways, addressed later in the paper.

Formalizing this intuition, we check whether the three partial configurations γ_a, γ_r, and $\gamma_{e\bar{c}}$ given by

(i) $\mathrm{supp}(\gamma_a) = \{a\}$ with $\gamma_a(a) = \mathtt{true}$,
(ii) $\mathrm{supp}(\gamma_r) = \{r\}$ with $\gamma_r(r) = \mathtt{true}$, and
(iii) $\mathrm{supp}(\gamma_{e\bar{c}}) = \{e, c\}$ with $\gamma_{e\bar{c}}(e) = \mathtt{true}$ and $\gamma_{e\bar{c}}(c) = \mathtt{false}$

are indeed feature causes of $\mathcal{E}$ w.r.t. $\mathcal{V}$ according to Definition 3.1: First, note that $[\![\gamma_{e\bar{c}}]\!] \cap \mathcal{V} = ([\![\gamma_a]\!] \cup [\![\gamma_r]\!]) \cap \mathcal{V} = \mathcal{E}$ and both, $[\![\gamma_a]\!] \cap \mathcal{V}$ and $[\![\gamma_r]\!] \cap \mathcal{V}$, are non-empty. Hence, **FC$_1$** is fulfilled for $\gamma_{e\bar{c}}$, γ_a, and γ_r. To check **FC$_2$**, we observe that $[\![\gamma_a\!\uparrow_a]\!] = [\![\gamma_r\!\uparrow_r]\!] = \Theta(F)$ and

$$[\![\gamma_{e\bar{c}}\!\uparrow_e]\!] \cap \mathcal{V} = \mathcal{E} \cup \{m, ms\}, \text{ and}$$
$$[\![\gamma_{e\bar{c}}\!\uparrow_c]\!] \cap \mathcal{V} = \mathcal{E} \cup \{mec, msec\}.$$

Hence, m is a counterfactual witness for γ_a, γ_r, and $\gamma_{e\bar{c}}$ w.r.t. a, r, and e, respectively, while mec can serve as such for $\gamma_{e\bar{c}}$ and c. It is easy to check that there are no further feature causes since for all other partial configurations sufficient for $\mathcal{E}$ w.r.t. $\mathcal{V}$ (**FC$_1$**) there are expansions towards γ_a, γ_r, or $\gamma_{e\bar{c}}$ (thus, violating **FC$_2$**).

3.1 Effect Properties and Sets

Our definition of feature causality relies on a given effect set, which is assumed to comprise all those valid configurations where the effect property holds. We now elaborate more on how to obtain effect sets from analyzing configurable systems. In fact, our generic definition supports a multitude of effect properties for which the only assumption is that there is an effective method to determine all configurations in which the effect property holds. Such methods also include variability-aware white-box analyses [88, 91], where the source code or operational behavior of system variants is accessible, as well as black-box analyses relying on testing or sampling [44, 53]. In the following paragraphs, we exemplify how to obtain effect sets from analysis results. Our discussions reflect the effect properties in the experimental evaluation section (see Section 5) and do not claim to be exhaustive.

Functional properties. To reason about causality w.r.t. functional properties, the effect set can be determined by variability-aware static analysis [10, 72] or model checking [5, 23, 68]. In the latter case, effect properties can be formalized, e.g., in a temporal logic such as LTL [69] or CTL [20]. Model checking configurable systems against LTL and CTL properties has broad tool support [e.g. 22, 25]. Given a formula φ that specifies the effect property, these tools explicitly return the effect set

$$\mathcal{E}_\varphi = \{\theta \in \mathcal{V} : \theta \models \varphi\}$$

of valid configurations $\theta \in \mathcal{V}$ whose corresponding system variants satisfy φ. Since model checking is based on an exhaustive analysis, an analysis also exposes those valid configurations for which the effect property does not hold. The same is possible for variability-aware static analysis [12, 72].

Non-functional properties. Besides functional properties, also non-functional properties of configurable systems can serve as effect property and give rise to an effect set. Let $\rho: \mathcal{V} \to \mathbb{R}$ be a function that results from a quantitative analysis of the configurable system in question. Values $\rho(\theta)$ for a valid configuration $\theta \in \mathcal{V}$

may stand for the performance achieved, the probability of failure, or the energy consumed in the system variant that corresponds to θ. To obtain ρ for real-world systems, Siegmund et al. [77, 78] presented a black-box method to generate linear-equation models for performance measures by multivariable linear regression on sampled configurations. Other black-box approaches rely on regression trees [44], Fourier learning [96], or probabilistic programming [27]. Related white-box approaches use insights of local measurements and taint analysis information [88] or profiling information [91]. Variability-aware probabilistic model checking pursues a white-box analysis approach on state-based operational models where effect properties are specified in quantitative variants of temporal logic [31, 84]. These approaches have been implemented in the tools ProFeat [19] and QFLan [86].

Given ρ that results from one of the analysis approaches mentioned above, an effect set can be specified by imposing a threshold $\tau \in \mathbb{R}$ combined with a comparison relation $\sim$ towards

$$\mathcal{E}_{\rho \sim \tau} = \{\theta \in \mathcal{V} : \rho(\theta) \sim \tau\}.$$

Example 3.3. In Example 3.2, we informally specified the effect of a "long decipher time" as taking more than three months to decrypt an email without having the encryption key available. By a variability-aware quantitative analysis on the email system, we may obtain a function ρ that, for a configuration θ, returns the minimal time in years to decipher an email sent with the system variant corresponding to θ. Analysis results could be, e.g., $\rho(\theta) = 0$ with no encryption, $\rho(\theta) = 10^{-7}$ with Caesar, $\rho(\theta) = 1$ with AES, and $\rho(\theta) = 2$ with RSA selected in θ, respectively. Then, $\mathcal{E}_{\rho > 0.25}$ provides the effect set $\mathcal{E}$ of Example 3.2.

On computing effect sets. The effect set and the set of valid configurations can be of exponential size in the number of features. An efficient computation of these sets depends on the analysis techniques used and are independent from our causal framework. However, specifically tailored variability-aware analysis techniques can tackle the exponential blowup, e.g., through symbolic representation of family models [29, 85].

3.2 Computation of Feature Causes

For a given effect set $\mathcal{E}$ and a set of valid configurations $\mathcal{V}$ along with a partial configuration ∂, Definition 3.1 directly provides a polynomial-time algorithm to decide whether ∂ is a cause of $\mathcal{E}$ w.r.t. $\mathcal{V}$ by checking $\mathbf{FC}_1$ and $\mathbf{FC}_2$. From this, we obtain a simple approach to compute the set Causes$(\mathcal{E}, \mathcal{V})$ by successively checking expansions for sets of features applied on elements in $\mathcal{E}$ as candidates for causes. Since there might be exponentially many such expansions, this approach easily renders infeasible already within a small number of features.

In this section, we present a practical algorithm to compute the set of causes, which relies on a connection of Definition 3.1 to the notion of prime implicants (see Section 2):

LEMMA 3.4. *For any partial configuration $\partial \in \Delta(F)$*

$$\llbracket \partial \rrbracket \cap \mathcal{V} \subseteq \mathcal{E} \quad iff \quad \llbracket \partial \rrbracket \subseteq (\Theta(F) \backslash \mathcal{V}) \cup \mathcal{E}.$$

Following this lemma, every cause of $\mathcal{E}$ w.r.t. $\mathcal{V}$ is also an implicant of $(\Theta(F) \backslash \mathcal{V}) \cup \mathcal{E}$ due to $\mathbf{FC}_1$ and even a prime implicant due to $\mathbf{FC}_2$. Conversely, every prime implicant ∂ of $(\Theta(F) \backslash \mathcal{V}) \cup \mathcal{E}$ for

Algorithm 1: Computation of feature causes

> **input** : $\mathcal{E}, \mathcal{V} \subseteq \Theta(F)$
> **output** : Causes$(\mathcal{E}, \mathcal{V})$

1 **if** $\mathcal{E} = \varnothing$ **then return** $\varnothing$
2 $\mathcal{P} := \text{COMPUTE-PRIMES}\big((\Theta(F) \backslash \mathcal{V}) \cup \mathcal{E}\big)$
3 **forall** $\partial \in \mathcal{P}$ *where* $\llbracket \partial \rrbracket \cap \mathcal{E} = \varnothing$ **do** $\mathcal{P} := \mathcal{P} \backslash \{\partial\}$
4 **return** $\mathcal{P}$

which $\llbracket \partial \rrbracket \cap \mathcal{E} \neq \varnothing$ is a cause due to $\mathbf{FC}_1$. This directly suggests an algorithm to compute causes via prime implicants: Algorithm 1 first generates prime implicants as cause candidates and then removes those candidates that are not sufficient for $\mathcal{E}$ w.r.t. $\mathcal{V}$. Figure 2 reflects this situation where γ and ∂ are prime implicants with γ being a cause and ∂ not: at least one effect instance is covered by γ, while this is not the case for ∂ and hence would be removed by Algorithm 1. Prime implicants of a set of configurations can be computed in polynomial time in the size of the input set [83], which directly leads to:

THEOREM 3.5. *Given valid configurations $\mathcal{V} \subseteq \Theta(F)$ and effect set $\mathcal{E} \subseteq \mathcal{V}$, Algorithm 1 computes Causes$(\mathcal{E}, \mathcal{V})$, the set of feature causes for $\mathcal{E}$ w.r.t. $\mathcal{V}$, in polynomial time in $|\Theta(F)|$.*

Note that the set of valid configurations and the effect set can be both exponential in the number of features and there might be exponentially many prime implicants [16] in the worst case. Hence, Algorithm 1 is exponential in the number of features.

Example 3.6. Let us illustrate the computation of feature causes of Example 3.2 by Algorithm 1. First notice that

$$(\Theta(F) \backslash \mathcal{V}) \cup \mathcal{E} = \Theta(F) \backslash \{m, mec, ms, msec\}$$

comprising 60 feature configurations. The prime implicants for this set are computed in Line 2, which yields

$$\mathcal{P} = \{\gamma_{\bar{m}}, \gamma_a, \gamma_r, \gamma_{e\bar{c}}, \gamma_{\bar{e}c}\}.$$

Here, we used notations as in Example 3.2, e.g., supp$(\gamma_{\bar{e}c}) = \{e, c\}$, $\gamma_{\bar{e}c}(e) = \text{false}$, and $\gamma_{\bar{e}c}(c) = \text{true}$. Clearly, all configurations covered by $\gamma_{\bar{m}}$ or $\gamma_{\bar{e}c}$ are not valid and hence also no effects. Thus, they are removed in Line 3, leading to Causes$(\mathcal{E}, \mathcal{V}) = \{\gamma_a, \gamma_r, \gamma_{e\bar{c}}\}$.

4 CAUSAL EXPLICATIONS

Since the number of feature causes can be exponential in the number of features, a mere listing of all causes is neither feasible nor expedient for real-world software systems. This holds for both, humans that have to evaluate causal relationships in configurable systems, e.g., during software development, and machines that might use feature causes for further processing and reasoning.

In this section, we present and discuss several methods to compute *causal explications*, i.e., mathematical or computational constructs that arise from processing feature causes to provide useful causal representations and measures [8]. Explications are closely related to *explanations*, by which we mean human-understandable objects employed within an integrated system, e.g., in feature-oriented software development or in production-level deployments.

Our methods for computing explications rely on techniques from propositional logic and circuit optimization [61, 65], responsibility

and blame [17], and feature interactions [40]. They all take a global perspective on sets of feature causes rather than only considering single feature causes in isolation. In the following, we fix sets of valid configurations $\mathcal{V} \subseteq \Theta(F)$ and effects $\mathcal{E} \subseteq \mathcal{V}$.

4.1 Distributive Law Simplification

A rather natural explication for a set of causes $C \subseteq \text{Causes}(\mathcal{E}, \mathcal{V})$ is to represent C as propositional logic formula, e.g., as the *characteristic formula* $\chi(C)$ defined in disjunctive normal form (DNF)

$$\chi(C) \;=\; \bigvee_{\partial \in C} \Big(\bigwedge_{\substack{x \in \text{supp}(\partial) \\ \partial(x)=\text{true}}} x \;\wedge\; \bigwedge_{\substack{x \in \text{supp}(\partial) \\ \partial(x)=\text{false}}} \neg x \;\Big).$$

Clearly, $\chi(C)$ has the same size as C and its representation does not exhibit any advantage compared to C. Methods to minimize propositional logic formulas [48, 60, 61] could be used to yield small formulas φ covering the same configurations as C, i.e., where $[\![\varphi]\!] = [\![C]\!]$. While beneficial for related problems in configurable systems analysis, e.g., for presence condition simplification [89], such methods vanish causal information, i.e., the set of causes C cannot be reconstructed from the reduced formula φ. To provide a small formula that maintains the causal information of C, we use a simple yet effective reduction method, which we call *distributive law simplification (DLS)*. The basic idea is to factorize common feature selections in a DNF formula step by step, exploiting the n-ary distributive law $[\![\bigvee_{i=1}^{n}(\phi \wedge \psi_i)]\!] = [\![\phi \wedge \bigvee_{i=1}^{n} \psi_i]\!]$. Each factorization leads to a length reduction of $(n-1) \cdot |\phi|$, where $|\phi|$ is the length of the propositional formula factored out. Obviously, these transformations are reversible, such that the original DNF $\chi(C)$ and hence the set of causes C can be reconstructed. The final formula length depends on the formulas factored out, the subformulas, and the factorization order. Determining a formula through DLS that has minimal size is close to global optimization problems for propositional logic formulas and thus computationally hard. For practical applications, we hence employ a heuristics that reduces a given formula φ in DNF by stepwise factoring out literals that have maximal number of occurrences in DNF subformulas. We denote the reduced formula obtained by this heuristics by $\text{DLS}(\varphi)$.

4.2 Cause–Effect Covers

The complete set of causes may contain several candidates to describe reasons for the effect emerging in a single system variant. If not interested in all causes but in a set of causes that covers all effects (i.e., that contains, at least, one cause for every system variant) we might ask for a preferably small *covering set*. Formally, a *cause–effect cover* of $\mathcal{E}$ is a set $C \subseteq \text{Causes}(\mathcal{E}, \mathcal{V})$ where $\mathcal{E} \subseteq [\![C]\!]$. We say that such a cover C is *minimal* if there is no cause–effect cover C' of $\mathcal{E}$ where $|C'| < |C|$.

Example 4.1. For the email system in Example 3.2 we directly see that $\{\gamma_{e\bar{c}}\}$ and $\{\gamma_a, \gamma_r\}$ are the only cause–effect covers of $\mathcal{E}$. Thus, $\{\gamma_{e\bar{c}}\}$ is a minimal cause–effect cover of $\mathcal{E}$ as $|\{\gamma_a, \gamma_r\}| > |\{\gamma_{e\bar{c}}\}|$.

Determining minimal cause–effect covers is computationally hard for the same reasons as for minimal prime–cover computation [65]. Hence, for practical applicability, heuristics that lead to nearly minimal cause–effect covers are of interest.

Definition 4.2. The binary relation $\trianglelefteq \subseteq \Delta(F) \times \Delta(F)$, where $\partial \trianglelefteq \partial'$ stands for ∂' to be *at least as general as* ∂ w.r.t. $\mathcal{E}$, is defined by

$$\partial \trianglelefteq \partial' \;\; \text{iff} \;\; [\![\partial]\!] \cap \mathcal{E} \subseteq [\![\partial']\!] \cap \mathcal{E}.$$

The set of *most general causes* $m\text{Causes}(\mathcal{E}, \mathcal{V})$ comprises those causes for $\mathcal{E}$ w.r.t. $\mathcal{V}$ that are $\trianglelefteq$-maximal in $\text{Causes}(\mathcal{E}, \mathcal{V})$.

From the above definition, we directly establish an algorithm to compute $m\text{Causes}(\mathcal{E}, \mathcal{V})$ by computing $\trianglelefteq$ on $\text{Causes}(\mathcal{E}, \mathcal{V})$ in quadratic time and selecting elements maximal w.r.t. $\trianglelefteq$. While $m\text{Causes}(\mathcal{E}, \mathcal{V})$ already provides a cause–effect cover of $\mathcal{E}$, there might be different most general causes that cover the same set of effect instances. Towards nearly minimal cause–effect covers, we thus might pick one of those candidates with minimal support to obtain an even further concise representation.

4.3 Responsibility and Blame

To measure the influence of causes on effects, Chockler and Halpern [17] introduced degrees of *responsibility* and *blame*, ranging from zero to one for "no" to "full" responsibility and blame, respectively. *Responsibility* measures how relevant a single cause is for an effect in a specific context. *Blame* denotes the expected overall responsibility according to a given probability distribution on all contexts where the effect emerges. We take inspiration from these measures and present corresponding notions for feature causality. In short, the degree of responsibility is the maximal share of features contributing to the effect, i.e., features that would have to be reconfigured to provide a counterfactual witness.

Example 4.3. We rephrase the *majority example* by Chockler and Halpern [17] in our setting. Consider eleven features whose configurations are all valid. We are interested in responsibilities for the effect that the majority of features is active. If all eleven features are active, each feature has a responsibility of $1/6$, since six features share the responsibility for the effect: Besides the feature of interest, further five features have to be reconfigured towards a majority of inactive features. In a configuration where six features are active and five not, each of the six active ones is fully responsible for the effect: If this feature would be reconfigured, more features would be inactive than active. We then assign responsibility of one to each of the six active features.

In what follows, we formalize degrees of responsibility and blame for single features as in the example above. An extension of this notion to partial configurations to explicate feature interactions is provided in Section 4.4.

Feature responsibility. Intuitively, the degree of responsibility of a single feature $x \in F$ is defined as the maximal share to contribute to causing the effect in an effect instance $\eta \in \mathcal{E}$. In the case that x does not appear in the support of any cause γ covering η, feature x does not contribute to causing the effect in η and thus has no responsibility. Otherwise, x shares its responsibility with, at least, a minimal number of other features, whose switch of its interpretation in η would lead to a counterfactual witness $\overline{\eta}$, i.e., $\overline{\eta} \in \mathcal{V} \backslash \mathcal{E}$. We formalize such switches in configurations by a function

$$\text{switch} \colon \wp(F) \times \Theta(F) \to \Theta(F)$$

where for any $Y \subseteq F$ and $\theta \in \Theta(F)$, we have $\text{switch}(Y, \theta)(y) = \theta(y)$ if $y \notin Y$ and $\text{switch}(Y, \theta)(y) = \neg\theta(y)$ otherwise.

Definition 4.4. The *degree of responsibility* of a feature $x \in F$ in the context $\eta \in \mathcal{E}$ for which there is $\gamma \in \text{Causes}(\mathcal{E}, \mathcal{V})$ with $\eta \in [\![\gamma]\!]$ and $x \in \text{supp}(\gamma)$ is defined as

$$\text{resp}(x, \eta) \;=\; \frac{1}{\min\{|Y| : Y \subseteq F, x \in Y, \text{switch}(Y, \eta) \in \mathcal{V} \backslash \mathcal{E}\}}$$

and as $\text{resp}(x, \eta) = 0$ otherwise.

Note that due to $\mathbf{FC}_2$ there exists at least one counterfactual witness in the case x appears in a cause covering η and hence, the denominator of the above fraction is finite and greater than zero.

Example 4.5. Continuing the email example (see Example 3.2), the mail feature m and sign feature s do not have any responsibility for a long decipher time in any configuration as they do not appear in any of the causes γ_a, γ_r, and $\gamma_{e\bar{c}}$. Also the AES feature a has no responsibility in configurations *mer* or *mers*, since the only covering causes are $\gamma_{e\bar{c}}$ and γ_r, not containing a in their support. Besides the analogous case for the RSA feature r, other degrees of responsibility are $1/2$: switching a feature x in η usually requires one further feature to switch towards a configuration of $\mathcal{V} \backslash \mathcal{E}$. For example, selecting the Caesar feature c in $\eta = mea$ requires also to deselect the AES feature a, leading to $\bar{\eta} = mec \in \mathcal{V} \backslash \mathcal{E}$.

Blame. Degrees of responsibility are locally defined w.r.t. a context, while one is surely also interested in a global measure of responsibility of a feature or partial configuration w.r.t. all possible contexts, i.e., effect configurations. The *degree of blame* is defined as the expected degree of responsibility on a probability distribution $\pi : \mathcal{V} \to [0, 1]$ over valid configurations $\mathcal{V}$, i.e., where $\sum_{\theta \in \mathcal{V}} \pi(\theta) = 1$.

Definition 4.6. The *degree of blame* of a feature $x \in F$ w.r.t. a distribution π over $\mathcal{V}$ is defined as

$$\text{blame}(x, \pi) \;=\; \sum_{\eta \in \mathcal{E}} \pi(\eta) \cdot \text{resp}(x, \eta).$$

Example 4.7. To illustrate the degree of blame, continue Example 4.5, where we assume a uniform distribution π over effect configurations, i.e., $\pi(\eta) = 1/|\mathcal{E}| = 1/4$ for all $\eta \in \mathcal{E}$ and 0 otherwise. Then, $\text{blame}(x, \pi) = 1/4 \sum_{\eta \in \mathcal{E}} \text{resp}(x, \eta)$ and thus, $\text{blame}(x, \pi)$ is $1/2$ for $x \in \{e, c\}$, $3/8$ for $x \in \{a, r\}$, and 0 for $x \in \{m, s\}$.

On the choice of blame distributions. The distribution π models the frequency of valid configurations to occur, for which there are several scenarios that lead to a reasonable definition of π. One natural distribution may model the frequency of users choosing a configuration in the configurable system. But also the frequency of effect configurations is useful, e.g., to model the frequency a certain bug is reported by users when the effect corresponds to the property of a malfunction. In the case such statistics are not at hand or one is interested in the degree of blame from a developer's perspective, uniform distributions over valid configurations or effects are canonical candidates for π.

4.4 Feature Interactions

A notorious problem in configurable systems is the presence of (inadvertent) *feature interactions* [3, 14], which describe system behaviors that emerge due to a combination of multiple features not easily deducible from the features' individual behaviors. The

detection, isolation, and resolution of feature interactions play a central role in the development of configurable systems and beyond [4, 94]. We now show how our black-box causal analysis at the level of features (see Section 3.1) can be used for the detection and isolation of feature interactions. These can provide the basis for fine-grained, white-box feature-interaction resolution as done by Garvin and Cohen [40].

Detection. The first problem we address is to detect the *necessity* of feature interactions for an effect to emerge. Garvin and Cohen presented a formal definition of *feature interaction faults* to capture faults in configurable systems that necessarily arise from the interplay between multiple features [40]. Notably, their characterization is also at the abstraction level of features and relies on black-box testing of faults, similar to our perspective on effects. We transfer their definition to our setting, but covering arbitrary effects instead of faults only. Recall that a partial configuration $\omega \in \Delta(F)$ is *sufficient* for $\mathcal{E}$ w.r.t. $\mathcal{V}$ if $\varnothing \neq [\![\omega]\!] \cap \mathcal{V} \subseteq \mathcal{E}$ (see $\mathbf{FC}_1$).

Definition 4.8. A partial configuration $\omega \in \Delta(F)$ is a *t-way interaction witness* for $\mathcal{E}$ w.r.t. $\mathcal{V}$ if

(FI$_1$) ω sufficient for $\mathcal{E}$ w.r.t. $\mathcal{V}$ with $|\text{supp}(\omega)| = t$ and
(FI$_2$) there is no $\hat{\omega}$ sufficient for $\mathcal{E}$ w.r.t. $\mathcal{V}$ with $|\text{supp}(\hat{\omega})| = t - 1$.

Basically, there is a one-to-one correspondence between interaction witnesses and feature causes with minimal support:

THEOREM 4.9. *For any partial configuration $\gamma \in \Delta(F)$ with $t = |\text{supp}(\gamma)|$ we have that*
(1) if γ is a t-way interaction witness for $\mathcal{E}$ w.r.t. $\mathcal{V}$, then $\gamma \in \text{Causes}(\mathcal{E}, \mathcal{V})$, and
(2) if $\gamma \in \text{Causes}(\mathcal{E}, \mathcal{V})$ and there is no $\hat{\gamma} \in \text{Causes}(\mathcal{E}, \mathcal{V})$ with $|\text{supp}(\hat{\gamma})| < t$, then γ is a t-way interaction witness for $\mathcal{E}$ w.r.t. $\mathcal{V}$.

To this end, Algorithm 1, in combination with a projection to feature causes with minimal support, can be used to decide whether the effect emerges necessarily from feature interactions: A necessary feature interaction takes place in the case these minimal feature causes all have a support involving, at least, two features.

Example 4.10. Returning to Example 3.2, there are exactly two causes that are both 1-way interaction witnesses: γ_a and γ_r. The cause $\gamma_{e\bar{c}}$ does not witness a necessary 2-way feature interaction since although having support size of two, γ_a and γ_r have support size one (cf. Theorem 4.9(2)). Hence, the effect describing long decipher time is not necessarily related to a feature interaction.

Isolation. The second problem we address is to pinpoint features responsible for feature interactions. For this, observe that Definition 4.8 is similar to Definition 3.1, but with a different notion of minimality: While $\mathbf{FI}_2$ ensures global minimality over all partial configurations, $\mathbf{FC}_2$ ensures local minimality through expansions, taking the individual selection of features into account. To pinpoint those features that actually interact towards the effect, feature causes can be interpreted as a form of *interaction witnesses* at the local level instead of the global perspective taken for *t*-way interaction witnesses: Switching some feature a cause does not ensure the effect to emerge anymore – hence, the switched feature is necessary for the effect. For instance, in Example 4.10, the interaction between encryption and Caesar being disabled is witnessed by the feature

cause $\gamma_{e\bar{c}}$. Feature causes thus also provide a criterion for feature interactions at the operational level and can be used to guide a more in-depth white-box feature interaction analysis, possibly reducing the naive exponentially-sized feature-interaction search space.

Feature interaction responsibility and blame. Any subset of the supports of feature causes that contains more than two features provides candidates for an actual interaction between the features contained. Since both, the number of feature causes and their expansions, can be exponential in the number of features, feature interactions isolated via a causal analysis might still be difficult to interpret by developers. Based on the degree of responsibility for single features, we now provide a variant to measure responsibility and blame of feature interactions, where high values indicate strong relevance of the interaction and low values weak relevance.

Definition 4.11. The *degree of responsibility* of a partial configuration $\partial \in \Delta(F)$ in the context $\eta \in \mathcal{E}$ for which there is $\gamma \in \text{Causes}(\mathcal{E}, \mathcal{V})$ with $\eta \in [\![\gamma]\!]$ and $\text{supp}(\partial) \subseteq \text{supp}(\gamma)$ and $\partial(x) = \gamma(x)$ for all $x \in \text{supp}(\partial)$ is defined as

$$\text{resp}(\partial, \eta) \;=\; \frac{1}{\min\{|Y| : Y \subseteq F, \text{supp}(\partial) \cap Y \neq \varnothing, \text{switch}(Y, \eta) \in \mathcal{V} \backslash \mathcal{E}\}}$$

and as $\text{resp}(\partial, \eta) = 0$ otherwise.

Note that, for $\partial(x) = \eta(x)$ and $\text{supp}(\partial) = \{x\}$, Definition 4.11 agrees with Definition 4.4. The degree of responsibility is non-zero in the case of a single feature if the feature appears in some cause, whereas the degree of feature interaction responsibility is non-zero if some cause is an expansion of the potential feature interaction. Feature interaction blame is defined as in Definition 4.6 but with replacing the single feature $x \in F$ by the partial configuration $\partial \in \Delta(F)$ standing for the potential feature interaction of interest.

5 EXPERIMENT SETUP

To evaluate our causal analysis and explication algorithms, we conducted a number of experiments comprising many analyses on community benchmarks and real-world examples from the area of configurable software systems.

5.1 Research Questions

Our evaluation is driven by four research questions that address the key issue of whether and how the notion of feature causality facilitates identifying root causes, estimating the effects of features, and detecting feature interactions in controlled and practical settings.

(RQ$_1$) Can feature causes be effectively computed in real-world settings and support the detection of reasons for different effects of interest?

(RQ$_2$) Do DLS representation, cause–effect covers, and responsibility and blame degrees provide concise causal explications?

(RQ$_3$) Is feature causality beneficial for guiding the configuration of systems under variability-aware constraints?

(RQ$_4$) Can feature interactions and configuration-dependent anomalies be detected and isolated based on feature causality?

5.2 Implementation

We implemented our algorithms to compute feature causes and explications in the prototypical tool FeatCause. Written in Python,

our tool relies on the engines for logical expressions and binary decision diagrams (BDDs) of PyEDA, a library for electronic design automation [28]. The tool takes the sets of valid feature configurations $\mathcal{V}$ and effects $\mathcal{E}$ as input. FeatCause supports different input formats for these sets, e.g., by Boolean expressions in DNF or CNF. Internally, sets of (partial) feature configurations are efficiently represented as reduced ordered BDDs [13]. In addition to their compact and hence space-efficient representation, we chose BDDs because they provide an efficient method to check satisfiability (required, e.g., for Line 3 in Algorithm 1). We first implemented a naive algorithm directly checking the conditions $\mathbf{FC}_1$ and $\mathbf{FC}_2$, but even for small examples, we easily ran into timeouts. Hence, we devised Algorithm 1, which uses prime implicants to efficiently determine feature causes (cf. Section 3). To compute prime implicants, we used the tool Espresso [61], well known from circuit optimization, through an interface that mediates between our BDD representations and the DNF representations in Espresso's PLA format. This interface is also used to provide minimal and nearly minimal cause–effect covers through Espresso-Signature and Espresso, respectively, which can then be compared with our heuristic cause–effect cover by most general causes (see Section 4.2). While it is well known that the length of DNFs can be exponential in the size of the BDD representing the same Boolean function, generating DNFs from our BDD representations did not face any significant blowup and required negligible time in all our experiments. Besides the core tool, we have implemented several conversion scripts to generate valid feature configuration sets from TVL [21] and effect sets from analysis results returned by variability-aware analysis tools such as ProVeLines [25] and ProFeat [19] (see also Section 3.1) and the data sets from Siegmund et al. [77] and Kaltenecker et al. [53].

5.3 Subject Systems

To answer our research questions, we selected a diverse set of subject systems, ranging from popular community benchmarks to more involved systems with non-functional properties and from real-world settings (see Table 1 for an overview).

From Cordy et al. [25], we use Minepump, Elevator, and CFDP systems and analyzed them against the accompanied LTL properties using the variability-aware model checker ProVeLines. Furthermore, we took the Email and Elevator from von Rhein et al. [89] and analyzed multiple defects provided as propositional logic formula generated by SPLVerifier [5].

For quantitative properties, we generated effect sets from configurable system analysis results as illustrated in Section 3.1 for three classes of systems. First, we considered configurable systems modeled for the variability-aware probabilistic model checker ProFeat [19], comprising a body sensor network (BSN) model [74] and a velocity control loop (VCL) model of an aircraft [32, 35]. In both systems, the reliability (R) of the system is analyzed in terms of the probability of failure of sensors and control components, respectively. Second, we generated effect sets from performance measurements of real-world configurable software systems that have been used to evaluate performance modeling techniques [53, 77]. In particular, we selected five systems from different domains: a compiler framework (LLVM), a database-system (BerkeleyDB), a compression tool (Lrzip), a video encoder (x264), and a toolbox for

Table 1: Statistics of feature causality experiments

Property [type]	System	#	$|\mathcal{V}|$	$|F|$	Time [s]	Average size of $\mathcal{E}$	C	mC	DLS
φ	CFDP	10	56	13	0.21	28	7.30	2.60	58%
	ELEVATOR₁	36	256	9	0.09	128	0.58	0.58	26%
	MINEPUMP	82	128	11	0.29	64	1.77	0.91	43%
ε_B	LINKEDLIST	42	204	19	18.97	102	47.40	8.36	54%
	LINUX	21	16 777 216	25	5.88	22	22.14	22.14	80%
	PKJAB	28	72	12	0.42	36	8.43	4.86	72%
	PREVAYLAR	42	24	7	0.20	12	3.57	2.17	93%
	SNW	42	3 240	36	2 751.32	1 620	2 110.31	18.90	37%
	ZIPME₁	42	64	9	0.19	32	4.19	2.69	93%
ε_M	CURL	28	768	14	83.07	384	96.75	67.39	59%
	H264	28	1 152	17	355.37	576	229.86	113.93	53%
	SQLITE	21	3 932 160	40	34 826.86	1 310 729	1 881.67	436.48	27%
	WGET	28	5 120	17	777.26	2 560	298.11	201.79	51%

Property [type]	System	#	$|\mathcal{V}|$	$|F|$	Time [s]	Average size of $\mathcal{E}$	C	mC	DLS
ε_T	APACHE	28	192	10	2.85	96	28.79	22.96	69%
	ELEVATOR₂	28	10	6	0.14	5	3.21	2.25	80%
	EMAIL	28	40	10	1.01	20	22.32	8.21	69%
	H264	56	1 152	17	32.24	576	68.98	22.71	51%
	ZIPME₂	42	640	16	1.73	320	11.07	9.98	72%
τ_R	BSN	4	298	11	0.63	149	21.00	8.75	68%
	VCL	22	2 097 152	21	60 403.78	1 048 576	3 718.05	3 718.05	36%
τ_T	BERKELEYDB	36	2 560	19	5.71	1 280	13.03	4.28	72%
	DUNE	46	2 304	32	3 950.15	1 152	724.87	46.00	46%
	LLVM	30	1 024	12	16.21	512	53.57	53.57	55%
	LRZIP	48	432	20	5.71	216	13.85	1.90	69%
	x264	48	1 152	17	2.38	576	7.50	3.75	78%

We list for each type of effect property (functional LTL properties φ, precision accuracy threshold properties ε, and other threshold properties τ) the considered subject systems, the numbers of experiments (#), valid configurations ($|\mathcal{V}|$), features ($|F|$), and the overall time in seconds to compute feature causes and most general causes. The second part of the table lists the average sizes of the effect sets ($\mathcal{E}$), feature causes ($C = \text{Causes}(\mathcal{E}, \mathcal{V})$), cause–effect covers by most general causes ($mC = m\text{Causes}(\mathcal{E}, \mathcal{V})$), and DLS formulas relative to the characteristic formula of causes (DLS=$|\text{DLS}(\chi(C))|/|\chi(C)|$ in percent).

solving partial differential equations (DUNE). For these systems, we chose thresholds on the runtime (T) of the system executions. Third, we constructed effect sets for several systems from studies on performance prediction of non-functional properties [78, 79, 81], such as APACHE, LINUX, SQLITE, and WGET. For these, we used the black-box approach by Siegmund et al. [80], which uses multivariable linear regression methods to generate variability-aware performance models. Our thresholds for constructing effect sets are imposed on the prediction accuracy of the following three different non-functional properties on those systems: runtime (T), binary size (B), and memory footprint (M).

5.4 Operationalization

For $\mathbf{RQ}_1$, we compute both sets $\text{Causes}(\mathcal{E}, \mathcal{V})$ and $\text{Causes}(\mathcal{V}\backslash\mathcal{E}, \mathcal{V})$, i.e., the feature causes of the effect property and its negation. Based on these feature causes, we further compute cause–effect covers $m\text{Causes}(\mathcal{E}, \mathcal{V})$ and $m\text{Causes}(\mathcal{V}\backslash\mathcal{E}, \mathcal{V})$, distributive law simplification (DLS), as well as responsibility and blame values for each single feature and cause to answer $\mathbf{RQ}_2$. For blame computations, we assume a uniform distribution over all effects, due to the absence of further statistical information and taking a developers' perspective (cf. Section 4.3). For $\mathbf{RQ}_3$, we compute single feature blames based on a uniform effect distribution to measure the influence of individual features onto the effect. We compute feature interaction blames on pairs of features to address $\mathbf{RQ}_4$, again assuming a uniform effect distribution. Table 1 provides key statistics about our experiments, focusing on model characteristics and the time to compute feature causes and most general feature causes. All experiments were conducted on an AMD Ryzen 7 3800X 8-Core system with 32GB of RAM running Debian 10 and Python 3.7.3.

6 RESULTS

We discuss our results w.r.t. the different kinds of causal explications of Section 4. First, we discuss statistics on our experiments, also quantitatively analyzing the potential of causal explications by most general causes and DLS-reduced formulas. Then, we address our research questions in more depth by means of three representative subject systems. Here, we focus on properties not detectable by classical causal white-box analysis methods [51, 95].

6.1 Descriptive Statistics ($\mathbf{RQ}_1$ and $\mathbf{RQ}_2$)

The computability of feature causes is of major interest for our evaluation. Table 1 provides an overview of the subject systems for which we generated effect sets and applied our feature causality analysis. We see that our algorithms compute feature causes in reasonable time, within a few seconds for most subject systems. To create the effect sets, we considered several effect properties, as described in Section 3.1: φ stands for LTL properties; ε_B, ε_M, and ε_T for thresholds on the accuracy of a prediction model for binary size, memory footprint, and runtime; and τ_R and τ_T for reliability and runtime thresholds, respectively. This variety of properties already illustrates the wide range of applications and potential use of feature causality. The sizes of valid configuration and effect sets crucially influence the time for computing feature causes, which is as expected since the complexity of Algorithm 1 is dominated by the computation of prime implicants of $(\Theta(F)\backslash\mathcal{V}) \cup \mathcal{E}$. Since our implementation relies on BDDs for the representation of valid configuration and effect sets, it is however well possible that within similar sizes, computation times can significantly differ. This is mainly due to the fact that BDD sizes highly depend on the specific nature of the represented Boolean functions and the variable order chosen [13]. For instance, while the experiments on DUNE and BERKELEYDB (see Table 1) have similar sized sets $\mathcal{V}$ and $\mathcal{E}$, their runtimes differ in two orders of magnitude. We see that the number of most general feature causes is often way smaller than the overall number of feature causes, which renders the creation of cause–effect covers by most general causes sensible to support concise explications. Interestingly, cause–effect covers by most general causes and ESPRESSO minimization [61] yield the same results in almost all of our experiments, which proves our heuristics to be effective. In the same vein, the application of DLS leads to great reductions of logical representations of feature causes, e.g., on average by almost 3/4 in the ELEVATOR₁ subject system (see Table 1).

> For $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we conclude that feature causes are computable in reasonable time. A substantial reduction of the set of feature causes and cause–effect covers can be performed with DLS formulas and most general feature causes, respectively.

Table 2: Minepump – (most general) feature causes

Blame	Feature cause (characteristic formula)
0.29	High ∧ Command ∧ ¬Stop ∧ MethaneAlarm
0.57	High ∧ Start ∧ MethaneAlarm
0.14	High ∧ Command ∧ ¬Stop ∧ MethaneSensor ∧ ¬MethaneQuery
0.57	High ∧ Start ∧ Stop
0.57	High ∧ Low ∧ Start
0.29	High ∧ Start ∧ MethaneSensor ∧ ¬MethaneQuery
0.29	High ∧ Low ∧ Command ∧ ¬Stop

6.2 Feature Cause Explications (RQ$_2$)

We discuss the explications of feature causes that we have generated by the example of the Minepump system [23], which is frequently used in the configurable systems' analysis community. This system models a water pump of a mine with $|F| = 11$ features on which requirements expressed in LTL are imposed (see Section 3.1). An analysis of the stabilization property, formalizing that the Minepump system eventually stabilizes (the pumps are eventually continuously on or off), using ProVeLines returned $|\mathcal{E}| = 28$ configurations where the property holds and $|\mathcal{V} \setminus \mathcal{E}| = 100$ configurations where it does not hold.

The direct interpretation of features responsible for this effect property is difficult, as it requires to investigate the result of all $|\mathcal{V}| = 128$ configurations. A causal analysis returned seven feature causes listed with their degree of blame in Table 2. They already provide hints which features are responsible for the property. Among the feature causes, three are most general, highlighted in Table 2. They have the highest degree of partial configuration blame while the most lengthy cause has the smallest degree. Our DLS heuristics on most general causes yields

$$\text{High} \wedge \text{Start} \wedge (\text{Stop} \vee \text{Low} \vee \text{MethaneAlarm})$$

providing a concise representation of feature cause candidates explicating the effect property. That is, selecting features High, Start, and one of the three features Stop, Low, or MethaneAlarm covers all causally relevant configurations for the Minepump system to stabilize. On other subject systems, explications are also effective, but with less drastic reductions than for the Minepump example.

Answering **RQ$_2$**, feature causes are of reasonable size compared to the complete analysis results, i.e., most general feature causes and DLS provide concise explications for feature causes. Responsibility and blame reflect the impact of feature causes.

6.3 Causality-guided Configuration (RQ$_3$)

Feature blame provides a quantitative measure on the causal impact of feature selections w.r.t. a set of configurations. This measure can be used to support configuration decisions, e.g., by prioritizing features with high blame values in case the effect property is desirable. We investigate such a causality-guided configuration on the velocity control loop (VCL) subject system [34, 35]. The VCL models an aircraft velocity controller in SimuLink for which its reliability in terms of probability of failure is of interest. A common principle to increase the reliability of a system is by *triple modular redundancy (TMR)* where system components are triplicated and their output are combined via a majority vote. Dubslaff et al. [32] suggested to

model and analyze systems with such *protection mechanisms* using family-based methods from configurable systems' analysis. To each component they assign a *protection feature* that specifies whether a component is triplicated or not. Comprising 21 components eligible for protection, the VCL model has $|\mathcal{V}| = 2^{21} = 2\,097\,152$ valid feature configurations. Clearly, the highest reliability is achieved by protecting all components. However, each protection comes at its costs in terms of execution time, energy costs, and packaging size. While it is known how to determine protection configurations with optimal reliability–cost tradeoff [32], reasons for why a protection configuration is optimal or why a component was selected for protection are typically unclear. We address this issue exploiting our causal analysis methods. Using the variability-aware probabilistic model checking tool ProFeat [19], we generated effect sets $\mathcal{E}_{\rho < \tau_R}$ w.r.t. ρ mapping to the probability of failure of the VCL within two control-loop executions and reliability thresholds τ_R between 0.019 and 0.064[1]. Table 3 shows the degree of feature blame for 18 protection features of the 21 components of VCL (cf. Section 4.3). The three components not shown in the table are input components, having zero degree of blame and hence do not contribute to the systems' reliability. With the lowest threshold $\tau_R = 0.019$, the effect set contains only 32 out of the 2^{21} feature configurations; almost all protections of components are responsible for the low probability of failure. Increasing the threshold lowers the degree of blame since the effect set increases, leading to less counterfactual witnesses. Using our blame analysis, we can directly deduce advice for engineers about components protections towards high reliability: With tight reliability constraints, one should protect the "Acceleration" component, followed by the "Integrator" component, as their blames are significantly higher than for other components (cf. upper rows of Table 3). When higher failure rates are acceptable, one should prefer to protect the components "Sum2" and "Sum3" instead of the "Integrator" component due to their higher impact on reliability (cf. lower rows of Table 3).

For **RQ$_3$**, we conclude that feature causes and degrees of blame reveal and quantify the impact of features on the desired effect and, this way, are able to guide the feature configuration process.

6.4 Feature Interactions (RQ$_4$)

Causal reasoning provides a new angle to study feature interactions in configurable systems. For illustration of how to detect and isolate feature interactions, we perform causal analysis on the Lrzip subject system, modeling a compression system for which runtime characteristics of the compression algorithms are of interest. Since the number of feature causes and their expansions can be both exponential in the number of features, a direct evaluation of the runtimes and causal analysis results is difficult. We hence investigate feature interactions through their degrees of blame as described at the end of Section 4.3. The subject effect sets $\mathcal{E}_{\rho > \tau_T}$ depend on the runtime $\rho \colon \mathcal{V} \to \mathbb{R}$ in seconds for a configuration compressing a file that is obtained as by Siegmund et al. [53, 77] and a runtime threshold τ_T

---

[1]To ensure timely analysis results, real-world failure probability measures were increased by a factor 100 [34]. Resulting higher values might seem unrealistic but arguably do not affect the causal analysis measuring the impact of protections.

Table 3: Feature blame for the VCL redundancy system

τ_R [10^{-2}]	MFunc	Sum1, Sum4	Memory	MOne. P, vCruise	SpentFuel, I	D	Prod	DragForce, EThrust	Integrator	Sum5	Sum3	Sum2	Acceleration
1.9	0.63	1.00	0.63	1.00	1.00	1.00	0.63	1.00	1.00	1.00	1.00	1.00	1.00
2.0	0.70	0.80	0.77	0.82	0.94	1.00	0.70	0.82	1.00	0.80	0.80	0.80	1.00
2.2	0.49	0.55	0.55	0.61	0.75	0.95	0.49	0.61	1.00	0.55	0.55	0.55	1.00
2.5	0.35	0.38	0.36	0.40	0.46	0.65	0.35	0.40	1.00	0.38	0.38	0.38	1.00
2.9	0.28	0.29	0.28	0.29	0.31	0.33	0.28	0.29	0.91	0.29	0.29	0.29	1.00
3.4	0.28	0.29	0.30	0.30	0.32	0.38	0.28	0.30	0.57	0.29	0.30	0.30	0.97
4.0	0.26	0.27	0.27	0.27	0.28	0.31	0.26	0.27	0.37	0.27	0.34	0.34	0.86
4.7	0.24	0.25	0.25	0.25	0.25	0.26	0.25	0.25	0.30	0.25	0.36	0.36	0.78
5.5	0.24	0.24	0.24	0.24	0.25	0.26	0.24	0.25	0.27	0.26	0.37	0.39	0.68
6.4	0.23	0.23	0.23	0.23	0.24	0.24	0.24	0.25	0.27	0.27	0.35	0.41	0.61

Table 4: Feature interaction blame for Lʀᴢɪᴘ

τ_T [10^2s]	Gzip 8	Gzip 9	Lrzip 4–6	Lrzip 7	Lrzip 8	Lrzip 9	Zpaq 4–7	Zpaq 8–9
2	0.002	0.009						
3			0.033	0.033	0.033	0.033		
4				0.042	0.042	0.042		
5				0.042	0.042	0.042		
6	1-way			0.042	0.042	0.042		
7	2-way			0.056	0.056	0.056	0.056	0.056
8				0.056	0.056	0.056	0.056	0.056
9				0.063	0.063		0.063	0.063
10				0.063	0.063		0.063	0.063
11				0.071			0.071	0.071
12							0.083	0.083
...								
22							0.083	0.083
23								0.250

(see end of Section 3.1). We then focus on 2-way interactions by investigating potential feature interactions between the compression algorithm and compression level responsible to have high runtimes. For this we compute degrees of feature interaction blame for partial configurations ∂ where $\text{supp}(\partial) \in \{Gzip, Lrzip, Zpaq\} \times \{1, \dots, 9\}$ and $\partial(x) = \texttt{true}$ for each $x \in \text{supp}(\partial)$. In the columns of Table 4, we show the degrees of feature interaction blames for thresholds τ_T ranging from $\tau_T = 200s$ to $\tau_T = 2\,300s$. Empty cells correspond to combinations of compression algorithm and level that do not appear in any cause and thus have zero blame. In these cases, we can conclude that no feature interaction takes place. Higher blame values indicate that the combined responsibility of the compression algorithm and level has a greater causal impact on runtime. Notably, we observe that, with an increasing threshold, the level of compression is increasingly responsible for longer runtime. Certain compression algorithms always have runtimes above the threshold independently of the compression level. This leads to a configuration blame of zero at any compression level, e.g., thresholds $\tau_T \leq 600s$ for the *Zpaq* algorithm shown in the upper right of Table 4. Note that, in these cases, *Zpaq* serves as 1-way interaction witness according to Definition 4.8. All greater thresholds for *Lrzip* and *Zpaq* do not have 1-way but 2-way interaction witnesses. That is, being above the runtime threshold is a result of a feature interaction between the compression algorithm and those compression levels not showing zero blame. Notice that the sums of the given feature interaction blames for $\tau_T \geq 700s$ that contain algorithms *Lrzip* or *Zpaq* add up to 1/2. That is, no other features are to be blamed for exceeding the runtime threshold.

Features have a dedicated meaning and one would hence expect higher runtimes for higher compression levels. To this end, it seems odd that the feature interaction of *Lrzip* and compression level 9 is less to blame for higher runtimes than for levels 7 and 8. This indicates an anomaly of the feature interaction between *Lrzip* and the compression levels 7, 8, and 9. Further investigations on analysis results and feature causes support these findings: averaged over all measurements of *Lrzip* configurations, we observe runtimes of 1 064.9s at compression level 7 (standard deviation 4.1s), 1 181.7s at level 8 (standard deviation 3.2s), and only 830.5s at level 9 (standard deviation 2.6s). Hence, *Lrzip* at level 9 is not causally relevant for exceeding the execution time threshold of 900s, as the compression level 9 feature is not contained in any cause with *Lrzip*. However, this insight is difficult to obtain relying purely on the performance

influence model given by ρ, as this would require handcrafted analysis of all 432 analysis results (see Table 1).

> For **RQ₄** we conclude that feature causes can provide hints for feature interactions and anomalies arising from them. Blame measures render themselves promising to quantify the influence of feature interactions that contribute to certain effects.

7 DISCUSSION

In this section, we discuss potential threats to validity of our experiments and relate our findings to existing work from the literature.

7.1 Threats to Validity

A threat to internal validity arises from the correctness of the analysis results from which we generated the effect sets. While for functional properties this threat is not crucial due to exact model-checking techniques used in our experiments, for non-functional properties the results have been partly established using machine learning. To mitigate this threat, we carefully chose effect-set thresholds such that the effect sets remain stable also within small threshold variations. Note that the choice of the effect set has no influence on the applicability of our causality definitions but only on to what extent causality can serve as an explication. For blame computations in our experiments, we assumed a uniform distribution over all effects, taking a developers' perspective where frequencies on how often an effect occurs in a real-world setting are not yet accessible. Other distributions could change our quantitative results, it is unlikely that they would alter our conclusions about causal influences of features and feature interactions. To increase the internal validity of our prototype, we implemented and evaluated several methods to compute causes. These include a naive brute-force approach and two additional methods to generate prime implicants, independent from the tool Esᴘʀᴇssᴏ.

Naturally, the choice of subject systems threatens external validity, which includes the kinds of effect sets on which we evaluate causality. To alleviate this threat, we included a wide variety of systems with multiple properties from different areas to our evaluation. They comprise several real-world software systems often used to evaluate sampling strategies and performance-modeling approaches. We further added several community benchmarks from the feature-oriented model-checking community as well as a large-scale redundancy system from reliability engineering.

7.2 Related Work

Various techniques for software defect detection have been proposed in the literature, ranging from testing [62] and static code analysis [64] to model checking [9]. These techniques have been also extended for analyzing configurable systems to tackle huge configuration spaces [85]. While such methods are able to identify defects and their location, the challenge of finding root causes for defects remains. A methodology to identify causes of defects during software development is provided through *root cause analysis* [73, 75], which can be supported by a multitude of techniques for causal reasoning [66, 67]. To the best of our knowledge, the foundations for a combination of configurable systems analysis and causal reasoning as we presented in this paper have not yet been addressed in the literature. In the following, we discuss related work in the fields of configurable systems analysis and causal reasoning.

Configurable systems analysis and explications. For analyzing configurable software systems, many approaches have been presented in the last two decades [70, 85]. There is broad tool support for variability-aware testing and sampling [10, 44, 52, 53, 80], static analysis [12, 72, 87, 88, 91], and model checking [5, 19, 23, 25, 68, 86].

There is a substantial corpus of work on determining those features in a configurable system that are responsible for emerging effects [e.g. 56, 71, 93]. The focus has been mainly on detecting feature interactions [3, 14, 15]. Siegmund et al. [78] and Kolesnikov et al. [55] describe non-functional feature interactions as interactions where the composed non-functional property diverges from the aggregation of the individual contributions of the single features. Garvin and Cohen [40] provided a formal definition of feature interaction faults based on black-box analysis to guide white-box isolation of interaction faults.

An incremental software configuration approach to optimize non-functional properties has been presented by Nair et al. [63], which complements our causality-guided software configuration we exemplified in Section 6.3 based on feature causality and pre-computed analysis results.

To reduce the size of propositional logic formulas in configurable systems, von Rhein et al. [89] proposed to exclude information about valid configurations and use two-level logic minimization, e.g., by the ESPRESSO heuristics [60, 61]. Our DLS method differs from this approach by prioritizing causal information over reduction.

Causal reasoning. Algorithmic reasoning about actual causes following the structural-equation approach by Halpern and Pearl [45, 46] is computationally hard in the general case [2, 38]. However, tractable instances such as the Boolean case have been identified by Eiter and Lukasiewicz [39]. For deciding whether a partial interpretation is an actual cause in the Boolean case, Ibrahim and Pretschner presented an approach based on SAT solving [49]. To compute all causes, their implementation relies on checking causality for all possible partial interpretations, suffering from an additional exponential blowup in the number of variables, which we avoid within our approach using prime implicant computations.

Using test generation methods relying on program trace information, program locations that are the origin of the defect can be identified [51, 76]. Analyzing differences between program states of sampled failing and passing executions, *delta debugging* identifies code positions relevant for an emerging failure [24]. Similarly, causes for detects can be determined by analyzing counterexample traces [11, 43]. Faults can be also located by causal inference on graphs constructed from statement and test coverage data [7].

Iqbal et al. present a static technique to generate causal models of a given configurable system using causal interference and statistical counterfactual reasoning [50]. This model is used to detect performance bugs and provide hints for their resolution. While we focused on actual causality and rigorous analysis, they are interested in *type causality* to answer more generic questions.

8 CONCLUDING REMARKS

Finding actual causes for an effect event becomes increasingly important in many research areas, also driven by high demands from politics and society. We introduced a formal definition and algorithms to identify causes in configurable systems that relies on counterfactual reasoning and connections to classical problems of propositional logics and circuit optimization. We demonstrated their potential by analyzing several subject systems, including real-world software systems and popular community benchmarks. To enable explanations for causes and their impact onto effects, we proposed explication techniques to concisely represent causes and quantify the causal impact of features. We showed that our explications are meaningful and can support the development of configurable software systems by causality-guided configuration and isolating feature interactions. With our prototypical implementation, we showed that our algorithms are effective on real-world systems of varying sizes and run in reasonable time.

While already shown to be effective, our implementation could be enhanced by directly integrating feature cause computations into optimized algorithms to compute prime implicants, e.g., relying on prime implicant computations at the level of BDDs [26]. Combined with state-of-the-art BDD libraries such as CUDD [82], the computation of causes might become feasible for even larger systems than considered in this paper.

Another direction is by enhancing our analysis of feature interaction blames (see Theorem 4.9) with in-depth white-box analyses [40] to pinpoint root causes in source code for a great variety of effect properties using feature causality.

Further applications could be imagined for *context-aware systems* where feature-oriented formalisms have been shown great applicability [18, 33, 59]. Here, our causal framework could reason about contexts responsible for certain effects, e.g., in self-adaptive systems [6]. In this vein, *dynamic configurable systems* [30, 41] are also an interesting direction to be considered. In such systems, features can be activated or deactivated during runtime, e.g., to model upgrade and downgrade of systems. The detection and isolation of feature interactions in dynamic configurable systems is a well-known challenge [58]. It is a promising avenue of further work to extend our causal framework to determine root causes and identify feature interactions in the dynamic setting [11].

REFERENCES

[1] Iago Abal, Jean Melo, Ștefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Transactions on Software Engineering and Methodology* 26, Article 10 (2018).

[2] Gadi Aleksandrowicz, Hana Chockler, Joseph Y. Halpern, and Alexander Ivrii. 2017. The Computational Complexity of Structure-Based Causality. *Artificial Intelligence Research* 58 (2017), 431–451.

[3] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. 2014. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Reports* 4 (2014), 1–24.

[4] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer.

[5] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 482–491.

[6] Uwe Aßmann, Christel Baier, Clemens Dubslaff, Dominik Grzelak, Simon Hanisch, Ardhi P P Hartono, Stefan Köpsell, Tianfang Lin, and Thorsten Strufe. 2021. *Tactile computing: Essential building blocks for the Tactile Internet.* Academic Press, Chapter 13, 301–326.

[7] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal Inference for Statistical Fault Localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 73–84.

[8] Christel Baier, Clemens Dubslaff, Florian Funke, Simon Jantsch, Rupak Majumdar, Jakob Piribauer, and Robin Ziemek. 2021. From Verification to Causality-Based Explications. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, Vol. LIPIcs:198. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 1–20.

[9] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking.* MIT Press.

[10] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2019. Static Analysis of Featured Transition Systems. In *Proceedings of the 23rd Systems and Software Product Line Conference (SPLC)*. ACM, 39–51.

[11] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. 2012. Explaining counterexamples using causality. *Formal Methods in System Design* 40 (2012), 20–40.

[12] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL$^{\text{LIFT}}$: statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th Conference on Programming Language Design and Implementation (PLDI)*. ACM, 355–364.

[13] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *Transactions on Computers* 35 (1986), 677–691.

[14] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41 (2003), 115–141.

[15] Muffy Calder and Alice Miller. 2006. Feature interaction detection by pairwise analysis of LTL properties—A case study. *Formal Methods in System Design* 28 (2006), 213–261.

[16] Ashok K. Chandra and George Markowsky. 1978. On the number of prime implicants. *Discrete Mathematics* 24 (1978), 7–11.

[17] Hana Chockler and Joseph Y. Halpern. 2004. Responsibility and Blame: A Structural-Model Approach. *Artificial Intelligence Research* 22 (2004), 93–115.

[18] Philipp Chrszon, Christel Baier, Clemens Dubslaff, and Sascha Klüppelholz. 2020. From Features to Roles. In *Proceedings of the 24th Systems and Software Product Line Conference (SPLC)*. ACM, Article 19, 11 pages.

[19] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2018. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing* 30 (2018), 45–75.

[20] Edmund M. Clarke, E. Allen Emerson, and Aravinda P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *Transactions on Programming Languages and Systems* 8 (1986), 244–263.

[21] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming* 76 (2011), 1130–1143.

[22] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2012. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer* 14 (2012), 589–612.

[23] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *Transactions on Software Engineering* 39 (2013), 1069–1089.

[24] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, 342–351.

[25] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2013. ProVeLines: a product line of verifiers for software product

[26] Olivier Coudert and Jean Christophe Madre. 1992. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proceedings of the 29th European Design Automation Conference (EURO-DAC)*. IEEE, 36–39.

[27] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2020. Mastering Uncertainty in Performance Estimations of Configurable Software Systems. In *Proceedings of the 35th Conference on Automated Software Engineering (ASE)*. IEEE, 684–696.

[28] Chris Drake. 2015. Pyeda: Data structures and algorithms for electronic design automation. In *Proceedings of the 14th Python in Science Conference (SciPy)*. 26–31.

[29] Clemens Dubslaff. 2019. Compositional Feature-Oriented Systems. In *Proceedings of the 17th Conference on Software Engineering and Formal Methods (SEFM)*, Vol. LNCS:12226. Springer, 162–180.

[30] Clemens Dubslaff. 2021. *Quantitative Analysis of Configurable and Reconfigurable Systems.* Ph. D. Dissertation. TU Dresden, Institute for Theoretical Computer Science.

[31] Clemens Dubslaff, Christel Baier, and Sascha Klüppelholz. 2015. Probabilistic Model Checking for Feature-Oriented Systems. *Transactions on Aspect-Oriented Software Development* LNCS:8989 (2015), 180–220.

[32] 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.

[33] Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan. 2019. Ontology-Mediated Probabilistic Model Checking. In *Proceedings of the 15th Conference on integrated Formal Methods (iFM)*, Vol. LNCS:11918. Springer, 194–211.

[34] Clemens Dubslaff, Andrey Morozov, Christel Baier, and Klaus Janschek. 2020. Iterative Variable Reordering: Taming Huge System Families. In *Proceedings of the 4th Workshop on Models for Formal Analysis of Real Systems (MARS)*, Vol. EPTCS:316. Open Publishing Association, 121–133.

[35] 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)*.

[36] Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel. 2021. FeatCause – Sources and Data. https://github.com/dubslaff/FeatCause

[37] Ellery Eells. 1991. *Probabilistic causality.* Cambridge University Press.

[38] Thomas Eiter and Thomas Lukasiewicz. 2002. Complexity results for structure-based causality. *Artificial Intelligence* 142 (2002), 53–89.

[39] Thomas Eiter and Thomas Lukasiewicz. 2006. Causes and explanations in the structural-model approach: Tractable cases. *Artifical Intelligence* 170, 6-7 (2006), 542–580.

[40] 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.

[41] Hassan Gomaa and Mohamed Hussein. 2003. Dynamic Software Reconfiguration in Software Product Families. In *Proceedings of the 5th Workshop on Software Profuct Family Engineering (PFE)*, Vol. LNCS:3014. 435–444.

[42] Irving J. Good. 1959. A theory of causality. *British Journal for the Philosophy of Science* 9 (1959), 307–310.

[43] Alex Groce and Willem Visser. 2003. What Went Wrong: Explaining Counterexamples. In *Proceedings of the 10th Workshop on Model Checking of Software*, Vol. LNCS:2648. Springer, 121–135.

[44] 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.

[45] Joseph Y. Halpern. 2015. A Modification of the Halpern-Pearl Definition of Causality. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI, 3022–3033.

[46] Joseph Y. Halpern and Judea Pearl. 2001. Causes and Explanations: A Structural-Model Approach - Part I: Causes. In *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann, 194–202.

[47] Joseph Y. Halpern and Judea Pearl. 2001. Causes and Explanations: A Structural-Model Approach - Part II: Explanations. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, 27–34.

[48] Edith Hemaspaandra and Henning Schnoor. 2011. Minimization for Generalized Boolean Formulas. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI/AAAI, 566–571.

[49] Amjad Ibrahim and Alexander Pretschner. 2020. From Checking to Inference: Actual Causality Computations as Optimization Problems. In *Proceedings of the 18th Symposium Automated Technology for Verification and Analysis (ATVA)*, Vol. LNCS:12302. Springer, 343–359.

[50] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2021. CADET: Debugging and Fixing Misconfigurations using Counterfactual Reasoning. arXiv:2010.06061 [cs.SE]

[51] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *Proceedings of the 42nd International*

Conference on Software Engineering (ICSE). ACM, 87–99.

[52] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The Interplay of Sampling and Machine Learning for Software Performance Prediction. *Software* 37 (2020), 58–66.

[53] 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.

[54] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.

[55] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in modeling performance of highly configurable software systems. *Software and Systems Modeling* 18 (2019), 2265–2283.

[56] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. 2004. Software fault interactions and implications for software testing. *Transactions on Software Engineering* 30 (2004), 418–421.

[57] David Lewis. 1973. Counterfactual Theories of Causation. *Journal of Philosophy* (1973), 556–567.

[58] Yu Liu and René Meier. 2009. Resource-Aware Contracts for Addressing Feature Interaction in Dynamic Adaptive Systems. In *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems (ICAS)*. IEEE, 346–350.

[59] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2016. Context Aware Reconfiguration in Software Product Lines. In *Proceedings of the 10th Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 41–48.

[60] Edward J. McCluskey Jr. 1956. Minimization of Boolean Functions. *Bell System Technical Journal* 35 (1956), 1417–1444.

[61] Patrick McGeer, Jagesh Sanghavi, Robert Brayton, and Alberto Sangiovanni Vincentelli. 1993. Espresso-Signature: A New Exact Minimizer for Logic Functions. In *Proceedings of the 30th Design Automation Conference (DAC)*. ACM, 618–624.

[62] Glenford J. Myers. 2004. *The art of software testing*. Wiley.

[63] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *Transactions on Software Engineering* 46 (2020), 794–811.

[64] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer.

[65] Wolfgang J. Paul. 1975. Boolesche Minimalpolynome und Überdeckungsprobleme. *Acta Informatica* 4 (1975), 321–336.

[66] Judea Pearl. 2009. *Causality: Models, Reasoning and Inference* (2nd ed.). Cambridge University Press.

[67] J. Peters, D. Janzing, and B. Schölkopf. 2017. *Elements of Causal Inference: Foundations and Learning Algorithms*. MIT Press.

[68] Malte Plath and Mark Ryan. 2001. Feature integration using a feature construct. *Science of Computer Programming* 41 (2001), 53–84.

[69] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science (SFCS)*. IEEE, 46–57.

[70] H. Post and C. Sinz. 2008. Configuration Lifting: Verification Meets Software Configuration. In *Proceedings of the 23rd Conference on Automated Software Engineering (ASE)*. IEEE, 347–350.

[71] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. 2008. Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 75–86.

[72] Alexander Von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *Transactions on Software Engineering and Methodology* 27, Article 18 (2018).

[73] ABS Group Inc. Risk and Reliability Division. 1999. *Root Cause Analysis Handbook: A Guide to Effective Incident Investigation*. Government Institutes.

[74] 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.

[75] James J Rooney and Lee N. Vanden Heuvel. 2004. Root Cause Analysis For Beginners. *Quality Progress* 37 (2004), 45.

[76] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating Failure Causes through Test Case Generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 309–319.

[77] 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.

[78] Norbert Siegmund, Sergiy S. 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)*. IEEE, 167–177.

[79] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. 2013. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology* 55 (2013), 491–507. Special Issue on Software Reuse and Product Lines.

[80] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal* 20 (2012), 487–517.

[81] Norbert Siegmund, Alexander von Rhein Sven, and Apel. 2013. Family-based Performance Measurement. *SIGPLAN Notices* 49, 3 (2013), 95–104.

[82] Fabio Somenzi. 1997. CUDD 3.0.0. http://vlsi.colorado.edu/~fabio/CUDD/html/

[83] Tadeusz Strzemecki. 1992. Polynomial-time algorithms for generation of prime implicants. *Journal of Complexity* 8 (1992), 37–63.

[84] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2016. Statistical Model Checking for Product Lines. In *Proceedings of the 7th Symposium on Leveraging Applications of Formal Methods (ISoLA)*, Vol. LNCS:9952. Springer, 114–133.

[85] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Computing Surveys* 47 (2014), 6:1–6:45.

[86] Andrea Vandin, Maurice H. ter Beek, Axel Legay, and Alberto Lluch Lafuente. 2018. QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems. In *Proceedings on the 22nd Symposium on Formal Methods (FM)*, Vol. LNCS:10951. Springer, 329–337.

[87] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems. *Automated Software Engineering* 27 (2020), 265–300.

[88] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE.

[89] 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.

[90] 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.

[91] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-Box Performance-Influence Models: A Profiling and Learning Approach. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE.

[92] Jon Williamson. 2009. Probabilistic Theories of Causation. *The Oxford Handbook of Causation* (2009), 185–212.

[93] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *Transactions on Software Engineering* 32 (2006), 20–34.

[94] Pamela Zave. 2001. Feature-Oriented Description, Formal Methods, and DFC. In *Proceedings of the Workshop on Language Constructs for Describing Features*. Springer, 11–26.

[95] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th Symposium on Foundations of Software Engineering (FSE)*. ACM, 1–10.

[96] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning (T). In *Proceedings of the 30th Conference on Automated Software Engineering (ASE)*. IEEE, 365–373.