On the Expressive Power of Languages for Static Variability

PAUL MAXIMILIAN BITTNER, Paderborn University and Ulm University, Germany
ALEXANDER SCHULTHEISS, Paderborn University and Bern University, Germany
BENJAMIN MOOSHERR, Ulm University, Germany
JEFFREY M. YOUNG, Input Output Global, USA
LEOPOLDO TEIXEIRA, Federal University of Pernambuco, Brazil
ERIC WALKINGSHAW, Unaffiliated, USA
PARISA ATAEI, Input Output Global, USA
THOMAS THÜM, Paderborn University and TU Braunschweig, Germany

Variability permeates software development to satisfy ever-changing requirements and mass-customization needs. A prime example is the Linux kernel, which employs the C preprocessor to specify a set of related but distinct kernel variants. To study, analyze, and verify variational software, several formal languages have been proposed. For example, the choice calculus has been successfully applied for type checking and symbolic execution of configurable software, while other formalisms have been used for variational model checking, change impact analysis, among other use cases. Yet, these languages have not been formally compared, hence, little is known about their relationships. Crucially, it is unclear to what extent one language subsumes another, how research results from one language can be applied to other languages, and which language is suitable for which purpose or domain. In this paper, we propose a formal framework to compare the expressive power of languages for static (i.e. compile-time) variability. By establishing a common semantic domain to capture a widely used intuition of explicit variability, we can formulate the basic, yet to date neglected, properties of soundness, completeness, and expressiveness for variability languages. We then prove the (un)soundness and (in)completeness of a range of existing languages, and relate their ability to express the same variational systems. We implement our framework as an extensible open source Agda library in which proofs act as correct compilers between languages or differencing algorithms. We find different levels of expressiveness as well as complete and incomplete languages w.r.t. our unified semantic domain, with the choice calculus being among the most expressive languages.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*; **Software configuration management and version control systems**; **Software product lines**.

Additional Key Words and Phrases: software variability, software product lines, configuration, expressiveness, language comparison

ACM Reference Format:
Paul Maximilian Bittner, Alexander Schultheiß, Benjamin Moosherr, Jeffrey M. Young, Leopoldo Teixeira, Eric Walkingshaw, Parisa Ataei, and Thomas Thüm. 2024. On the Expressive Power of Languages for Static

Authors' Contact Information: Paul Maximilian Bittner, paul.bittner@uni-ulm.de, Paderborn University and Ulm University, Germany; Alexander Schultheiß, alexander.schultheiss@hu-berlin.de, Paderborn University and Bern University, Paderborn, Germany; Benjamin Moosherr, benjamin.moosherr@uni-ulm.de, Ulm University, Ulm, Germany; Jeffrey M. Young, jeffrey.young@iohk.io, Input Output Global, Longmont, Colorado, USA; Leopoldo Teixeira, lmt@cin.ufpe.br, Federal University of Pernambuco, Pernambuco, Brazil; Eric Walkingshaw, eric@walkingshaw.net, Unaffiliated, Corvallis, Oregon, USA; Parisa Ataei, paris.ataei@gmail.com, Input Output Global, Buffalo, New York, USA; Thomas Thüm, thomas.thuem@uni-paderborn.de, Paderborn University and TU Braunschweig, Paderborn, Germany.

ACM 2475-1421/2024/10-ART307
https://doi.org/10.1145/3689747

Variability. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 307 (October 2024), 32 pages. https://doi.org/10.1145/3689747

1 Introduction

Explicit variability in software is a reoccurring phenomenon across many areas in science and business [63]. Operating systems such as the Linux kernel offer many configuration options to adapt to hardware or user needs, among other concerns [108]. Extensive variability is also common in other domains, such as file systems [100], cars [66], cloud systems [35, 145], robotics [67], mobile apps [86], bioinformatics [32], and satisfiability solving [130, 144] to name a few.

Variability means that a system or piece of information may emerge in similar but different *variants* from a common set of underlying atomic elements [7]. For example, the source code of the Linux kernel is variational because it can *and must* be configured to a particular kernel variant *before* a kernel can be used. While the Linux kernel consists of a single code base, its multiple thousand configuration options [61, 72, 127] impose many different kernel variants, in fact so many, that the number of variants cannot even be computed for newer versions [90, 127].

In research, variability is tackled by dividing its concerns into *problem* and *solution* space [7]. The problem space is concerned with specifying the set of available configuration options, parameters, or features, and documenting valid and invalid combinations. Specifying the set of valid feature combinations, referred to as *configurations*, reduces to a satisfiability problem [18]. As an example, in the Linux kernel, the problem space is implemented in terms of build and configuration files that govern which files to include and how conditional C preprocessor flags are allowed to be set to adhere to dependencies and conflicts [87, 110]. The solution space is concerned with implementing the functionality of each feature, and providing a mechanism to derive a variant's source code from its configuration. In the Linux kernel, the solution space is realized in terms of a code base annotated with C preprocessor statements, which are resolved by the C preprocessor, which derives a particular variant by preprocessing all annotations [87, 110].

To model and analyze solution space variability, numerous formal variability languages have been used [15, 27, 33, 47, 53, 64, 75, 77, 97]. These languages model variability following either the *annotative* or *compositional* paradigm [75]. Annotative languages embed variability annotations into a superimposed state of the variants, and derive variants by discarding all parts that belong to an excluded feature. For instance, many annotative languages depict variability in terms of a choice $F\langle e_{then}, e_{else}\rangle$ between alternative expressions e_{then} and e_{else}, where the annotation F denotes a feature whose inclusion or exclusion determines which alternative to pick [33, 53, 64, 136]. Choices are an abstract representation of if-then-else expressions and were, for example, successfully employed to type check all variants of the Linux kernel [76]. Compositional languages model variability in terms of distinct modules, each representing a unique feature. Modules can be composed in different ways, such as superimposing trees [11], weaving aspects [6], or loading plug-ins. For instance, feature structure trees [9, 11] model features as trees that can be composed by merging them in terms with a depth-first traversal. Within a Java graph library, a feature `Color : graph.Edge.color` named `Color` would add a `color` field to the `Edge` Java class, when imposed to the base feature `Base : graph.Edge.nodes`, yielding a syntax tree in which the `Edge` class has both the `color` and `nodes` fields.

With two paradigms for variability, a zoo of languages, and different syntactical constructs available, several questions remain unanswered. Which language should be picked for new research efforts or tools? Do research results based on one language also apply to other languages? Can all languages express the same sets of variational systems? To the best of our knowledge, a formal characterization and comparison of variability languages has not yet been attempted, and discussions remain scarce, mostly informal, or brief [75, 136, 137].

In this paper, we begin the journey to answer these questions by introducing a formal framework for comparing variability languages to guide researchers, practitioners, and language designers. We distill a common semantic domain for variability languages as a basis for comparisons and to capture a widely used intuition for static and explicit variability: a finite, non-empty set of system variants identified by configurations (i.e., feature (de-)selections). We then explore the three basic yet unexplored and neglected properties of completeness, soundness, and expressiveness. We consider a language complete iff it may describe any non-empty, finite set of variants, and hence serves as a general-purpose variability mechanism. For instance, if Linux's variability mechanism were incomplete, there would be kernel sets that could not be specified statically. Conversely, we consider a language sound iff every term describes a non-empty, finite set of variants, and hence all terms are valid. If Linux's variability mechanism were unsound, then a valid configuration could produce something else than a kernel implementation, such as a plain number or a sandwich recipe. Finally, expressiveness relates languages, where we say that a language L is at least as expressive as another language M iff L can describe any set of variants described by M. From a researcher's perspective, such comparisons bridge the gaps between parallel research efforts, increasing the impact of research results formulated in one particular language to now be useful to a more expressive language and its community, potentially even rendering the results independent from the languages they were formulated in. From a practitioners perspective, proofs of the above properties may come as correct compilers between languages or differencing algorithms, turning theory into practice.

As a case study, we begin charting the space of variability languages by formally comparing common variability languages and dialects within our framework. By instantiating these languages within our framework, we show that our formulation of a semantic domain fits these languages. We prove that there are complete and sound languages with respect to our semantic domain. By constructing respective compilers and proving them correct, we also show that the other languages can be translated to those complete languages; rendering existing research efforts and results compatible with each other. Interestingly, we found that there are incomplete languages in our semantic domain, where incompleteness stems from particular but reasonable design decisions. To also cover the widely adopted [4, 26, 27, 70, 75, 77, 97] but rarely formalized and understudied concept of optional variability in our case study, we introduce the option calculus as a minimal language for this purpose. We formalize our framework and our case study as an extensible open-source Agda library called Vatras [1]. Hence, our case study acts as a web of proven-to-be-correct compilers and differencing algorithms, which can be used out of the box, for example to apply techniques based on one language to another. In summary, we contribute:

Unified semantics by introducing a common semantic domain for variability languages, and by explicitly formalizing the semantics of some languages for the first time,

a formal framework for comparing the expressiveness, completeness, and soundness of variability languages to bridge the gap between parallel research efforts,

the option calculus as a formal variability language that is the first to rigorously formalize the widely adopted concept of optional variability,

a map of languages that formally clarifies the properties and relationship between existing, representative variability languages, and

an open-source Agda library that formalizes the framework and above contributions in a reusable library of compilers and differencing algorithms [1].

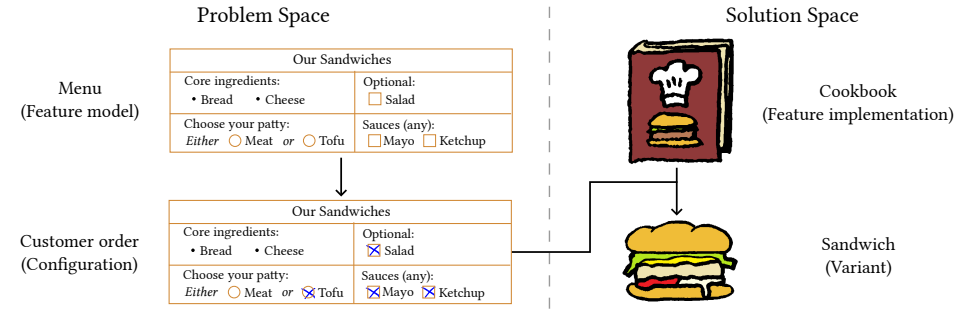


Fig. 1. Typical workflow for engineering variational systems at the example of a sandwich restaurant.

2 Preliminaries and Running Example

Suppose that you are the cook at a new sandwich diner that will open soon. From a market analysis, your supervisor distilled the most popular and affordable sandwiches into a menu shown in the top left of Figure 1. According to the specification, a sandwich *must* have bread and cheese, must have *either* tofu or meat as a patty, and can *optionally* have salad, while sauces can be freely combined. Each customer is handed a menu to freely configure a sandwich to their liking, for example with salad, all sauces and tofu as depicted at the bottom left of Figure 1.

Your task as the cook is to produce sandwiches according to the specification because sandwiches that do not satisfy the supervisor's requirements might sell badly or require unavailable ingredients. Therefore, you write a custom sandwich cookbook, shown in the top right of Figure 1, in which you specify how the ingredients can be combined to create sandwiches according to the requirements. When you are handed a customer's order, you can use the knowledge from your recipe book to produce the respective sandwich, shown at the bottom right of Figure 1.

Apart from sandwiches, variability of this kind may occur in various software or systems [63] and has mostly been addressed in the context of software product line engineering [7, 39, 113]. Explicitly designing variability has the goal of increasing reuse when developing a set of distinct but related products, usually referred to as *variants*, such as all the sandwiches that can be built according to the specification in Figure 1. Commonalities and differences between variants are typically expressed in terms of atomic configuration options, referred to as *features* [7, 39, 113], such as the sandwich ingredients. As emphasized by the left and right half in Figure 1, software product-line engineering distinguishes two spaces of variability:

The problem space specifies the names of features and the constraints among them in terms of a variability or feature model [7, 18, 73]. Essentially, a feature model denotes a satisfiability problem to decide whether a certain set of selections for features is valid [18, 40, 103]. In our example, the menu at the top left of Figure 1 denotes a feature model, which, for example, declares that exactly one patty must be chosen, and not zero or both. An assignment of features to selection values (e.g., Booleans or integers) is referred to as a *configuration* [7]. In our example, a configuration is given by a customer filling out the menu (bottom left in Figure 1).

The solution space refers to the implementation of features to derive variants from configurations. The main concern in the solution space is mapping features or their combinations to implementation artifacts. This mapping determines which artifacts should be included or excluded to generate a variant with a specific configuration. The cookbook in Figure 1 depicts a variational system that specifies how ingredient names (i.e., features) map to the ingredients  ,  ,  , and so on, and how ingredients have to be prepared to obtain a sandwich variant.

A prominent example of annotative solution space variability for software is the C preprocessor, one of the most widely used tools to implement explicit and static variability [93], such as in the Linux kernel [87]. A C code base with C preprocessor directives denotes *a set* of C programs. Only by running the preprocessor with a certain configuration, a particular program is obtained.

In this work, we focus on languages for solution space variability (i.e., cookbooks for sandwiches or any other domain of interest, such as programming languages). Both problem and solution space have a rich landscape of languages to encode their respective variational artifacts. Problem space languages have been formally and informally compared on different levels of abstraction, as we discuss in Section 6. In contrast, formal solution space languages have not yet been compared, in particular not in a formal way. For brevity, we just write *variability language* to refer to languages for solution space variability for the remainder of this paper.

Notation. We use syntax highlighting: constants are violet, grammar symbols are green (i.e., inductive constructors), names of sets are blue, and relations and functions are magenta. We represent lists of values $a \in A$ of some set A, as tuples $(a_1, \ldots, a_n) \in A^n$ as common in set theory. For some recursive functions on lists though, we rely on an inductive notation from type theory and functional programming (e.g., Haskell, Agda), where $[\,]$ represents the empty list and $a :: l$ prepends a value $a \in A$ to a list $l \in A^m$. For example, $(1, 2, 3, 4)$ denotes the same list as $1 :: 2 :: 3 :: 4 :: [\,]$. We abbreviate singleton lists $a :: [\,]$ as $[a]$. We assume $0 \notin \mathbb{N}$ and use $\circ$ to denote function composition.

3 An Overview on Formal Variability Languages

To model, analyze, and study variability, a range of formal languages were proposed, coming in the forms of calculi [53, 64, 136], algebras [11], graphs [16, 27, 33, 49], or generic trees [74, 97]. These languages model variability as a general, reoccurring phenomenon not fixed to a particular technology, such as the C preprocessor. In this section, we give an overview about common formal languages for variability and briefly introduce and unify their syntax and semantics.

3.1 Semantics

To understand, model, and compare variability languages, it is essential to agree on a common semantic domain, that answers the question: What is static variability? A short answer is that most languages have the common goal to describe a set of related but different *variants* of a system for a certain range of artifacts such as source code or build files [7, 39].

Before we can model variability within systems, we first need a generic model for systems *without* variability. Most variability languages use some kind of generic tree [11, 16, 27, 33, 53, 64, 74, 75, 97]. Trees cover concrete and abstract syntax of formal languages, and hence basically any computer-readable language, including programming languages and any sequential data (e.g., lines of text). Hence, we model a variant as a tree, where each node contains a single piece of atomic data $a \in A$ of any granularity. Crucially, a variant represents *no* variability.

Definition 3.1 (Variant). A variant $v \in \mathbb{V}(A)$ is a tree of atoms $a \in A$, where $\mathbb{V}(A)$ denotes the set of all variants over a set of atoms A. We denote variants as expressions $\underline{e}$ with the following syntax:

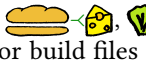
$$\underline{e} \quad ::= \quad a\text{-}\prec \underline{e}, \ldots, \underline{e} \succ\text{-}$$

where $\underline{e}, \ldots, \underline{e}$ denotes a finite and potentially empty list of sub-expressions. While a is an atom, we refer to a production $a\text{-}\prec \ldots \succ\text{-}$ as an *artifact* to distinguish the contained data a from the node $\prec \ldots \succ$ holding that data. For convenience, we write a instead of $a\text{-}\prec\succ\text{-}$ for leaf artifacts.

Example 3.2 (Atoms and Variants). In our running example in Section 2, we used sandwiches, where ingredients, such as  ,  , and  are atoms and the composition of ingredients denotes

Language		Syntax e	Configuration Language C	Semantics $[\![.]\!] : e \to C \to \underline{e}$
Core Choice Calculus (CCC)	[53, 136]	$e ::= a\prec e,\dots,e\succ \mid D\langle e,\dots,e\rangle$	$\mathbb{F} \to \mathbb{N}$	$[\![a\prec e_1,\dots,e_n\succ]\!](c) := a\prec [\![e_1]\!](c),\dots,[\![e_n]\!](c)\succ$ $[\![D\langle e_1,\dots,e_n\rangle]\!](c) := [\![e_{min(c(D),n)}]\!](c)$
Binary Choice Calculus (2CC)	[many]	$e ::= a\prec e,\dots,e\succ \mid D\langle e,e\rangle$	$\mathbb{F} \to \mathbb{B}$	$[\![a\prec e_1,\dots,e_n\succ]\!](c) := a\prec [\![e_1]\!](c),\dots,[\![e_n]\!](c)\succ$ $[\![D\langle l,r\rangle]\!](c) := \begin{cases} [\![l]\!](c), & c(D)=true, \\ [\![r]\!](c), & \text{else} \end{cases}$
Algebraic Decision Trees (ADT)	[16, 33]	$e ::= \text{leaf } \underline{e} \mid D\langle e,e\rangle$	$\mathbb{F} \to \mathbb{B}$	$[\![\text{leaf } v]\!](c) := v$ $[\![D\langle l,r\rangle]\!](c) := \begin{cases} [\![l]\!](c), & c(D)=true, \\ [\![r]\!](c), & \text{else} \end{cases}$
Gruler's Language (GL)	[64]	$e ::= \text{ntrl} \mid \text{asset } a \mid e \parallel e \mid e \oplus_{n\in\mathbb{N}} e$	$\mathbb{N} \to \mathbb{B}$	$[\![\text{ntrl}]\!](c) := \varepsilon$ $[\![\text{asset } a]\!](c) := a$ $[\![l \parallel r]\!](c) := [\![l]\!](c) \parallel [\![r]\!](c)$ $[\![l \oplus_n r]\!](c) := \begin{cases} [\![l]\!](c), & c(n)=true, \\ [\![r]\!](c), & \text{else} \end{cases}$
Option Calculus (OC)	new	$e ::= a\prec t,\dots,t\succ$ $t ::= e \mid O\langle t\rangle$	$\mathbb{F} \to \mathbb{B}$	$[\![a\prec e_1,\dots,e_n\succ]\!](c) := a\prec\kappa([\![e_1]\!](c),\dots,[\![e_n]\!](c))\succ$ $[\![O\langle e\rangle]\!](c) := \begin{cases} [\![e]\!](c), & c(O)=true, \\ \varepsilon, & \text{else} \end{cases}$
Feature Structure Trees (FST)	[9, 11]	$e ::= a \blacktriangleleft f,\dots,f \blacktriangleright$ $f ::= F : \underline{e},\dots,\underline{e}$	$\mathbb{F} \to \mathbb{B}$	$[\![a \blacktriangleleft (F_1 : fs_1),\dots,(F_n : fs_n) \blacktriangleright]\!](c) := a\prec \bigoplus_{\substack{i\in\{1,\dots,n\},\\ c(F_i)=true}} fs_i \succ$

Table 1. Profiles of prominent formal variability languages

a variant of a sandwich, such as a sandwich with cheese and salad  . Common target systems for static variability are source code, documentation, or build files as implemented by the C preprocessor [87, 93], KConfig [52], autoconf [131], or GNU M4 [132]. When representing source code as lines of text, as done by many tools, the set of atoms A is the set of all lines of text, and a text file could be represented as a variant $line_1\prec line_2\prec\dots\prec line_n\succ\succ\succ$. Alternatively, variants can represent concrete or abstract syntax trees of formal languages in which case atoms A corresponds to the names of production rules and tokens of a grammar (i.e., labels of nodes in the syntax tree) [74]. For instance, an arithmetic expression $1+5\cdot 7$ can be represented as $+\prec 1, \cdot\prec 5, 7\succ\succ$.

3.2 Overview

Table 1 presents a selection of variability languages from the literature. For each language, we cite its origin work(s) and show its syntax, configuration language C, and semantics $[\![.]\!] : e \to C \to \underline{e}$, which is a function that configures an expression e with a configuration $c \in C$ of the respective configuration language to a variant $\underline{e}$. There exist more variability languages than we can cover in a single paper, so we distilled our overview based on three criteria. First, we favor maturity of formalization (i.e., whether syntax or semantics are formalized). Second, we favor genericity (i.e., whether the language models variability as a general-purpose phenomenon and not tied to a particular use case or technology). Third, we favor languages that are representative for similar dialects or languages that might target more specific use cases.

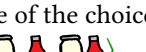
We streamlined and simplified the definition of syntax or semantics for some of the languages in Table 1. Most languages were developed independently from each other and thus use different notation or theories to express similar concepts. For syntax, we decided for a representation that is as simple as possible while retaining the spirit of the original language. For example, algebraic decision diagrams are defined as graphs, which alternatively can be formalized via the given

grammar. Moreover, the languages often rely on slightly different but equivalent semantic domains (basically generic trees). For some languages, the semantics were not even formalized at all, given only in natural language, examples, or implementations. We hence unify semantics here to have the same signature $e \rightarrow C \rightarrow \underline{e}$ for comparability, and to adhere to adjusted syntax.

3.3 Choice Calculus (CCC and 2CC)

The choice calculus was created to serve a role analogous to the lambda calculus for programming languages but for modeling and analyzing static variability [53, 136]. The idea is to annotate a variant $\underline{e}$ by embedding choices $D\langle e_1, \ldots, e_n\rangle$, $n \geq 1$ where D is referred to as a dimension and the sub-expressions e_i are referred to as alternatives. A choice denotes the necessity to choose exactly one alternative e_i. The dimension $D \in \mathbb{F}$ is a name to identify the choice and to convey the choice's meaning to potential users, where $\mathbb{F}$ is a set of names (e.g., $\mathbb{N}$, text, or feature names according to a feature model, cf. Section 2). Based on our running example from Section 2, the expression

(1)

encodes sandwiches by embedding choices into a variant expression (cf. Definition 3.1). The outer artifact  denotes that a sandwich always has bread at the root of the expression. The choices for salad, patties, and sauces are named by the Salad, Patty, and Sauce dimensions, respectively. Within bread, a sandwich (1) maybe has salad, denoted by the choice Salad$\langle$  , $\varepsilon\rangle$ between salad and an "empty" ingredient ε, (2) always has cheese because  is not nested in a choice, (3) can have either meat or tofu because of the choice Patty$\langle$  $\rangle$, and (4) can have mayonnaise and/or ketchup or none: Sauce$\langle\varepsilon$,  $\rangle$.

To obtain a variant for a given configuration, the semantics $[\![.]\!]$ eliminate all choices from an expression, leaving only artifacts $a\prec \ldots \succ$ and thus a variant $\underline{e}$. Therefore, a configuration $c : \mathbb{F} \rightarrow \mathbb{N}$ determines which alternative to pick for each choice by mapping each unique dimension to a natural number. While the denotational semantics of the choice calculus have been formalized in different ways [53, 136, 137], we follow the functional style of creating a *variant generator function* for an expression [137] for brevity in presentation and proofs. Hence, the semantics $[\![e]\!]$ of an expression $e \in$ CCC is a function $[\![e]\!] : (\mathbb{F} \rightarrow \mathbb{N}) \rightarrow \underline{e}$, which maps each configuration $c : \mathbb{F} \rightarrow \mathbb{N}$ to a variant $\underline{e}$. In particular, an artifact $a\prec e_1, \ldots, e_n\succ$ does not denote variability and hence must not be configured. We thus keep the atom a but configure all sub-expressions $e_1, \ldots, e_n$, which potentially contain choices. A choice $D\langle e_1, \ldots, e_n\rangle$ denotes variability that must be resolved. We do a lookup in the configuration c for the dimension D and pick the chosen alternative e_i, where $i = min(c(D), n)$ can at most be the number of available alternatives n.[1] Since choices might have any *arity* (i.e., number of alternatives) configurations cannot be easily restricted to yield indices in bounds. We will explore restricting the arity to be in bounds by design later in Section 5.

As an example, we can configure our sandwich from Expression 1. Let's say we want salad, tofu, and ketchup and define a configuration c accordingly as c(Salad) = c(Patty) = 1, c(Sauce) = 3. We then obtain our desired sandwich via the semantics: $[\![\text{Expression 1}]\!](c) =$  . The result contains no choices and therefore is a variant.

To simplify reasoning, choices are frequently restricted to be binary, such as for variability-aware syntax- [78] and type-checking [36, 79, 94] variants of C preprocessor-based software, including the Linux kernel, and many other use-cases [14, 31, 37, 126, 137, 143, 144]. Following established naming conventions [136], we refer to the choice calculus with choices of any arity as the *core*

[1] Resolving dimensions via natural numbers is a simplification of the original works on choice calculus [53, 136, 137]. There, a dimension D identifies each of its alternatives with a *tag* $D.t$, and a configuration maps each dimension to one of its tags. For brevity in presentation and proofs, we chose to replace tags by an equivalent representation via natural numbers, analogous to how *de Bruijn indices* simplify formal reasoning on the lambda calculus [45].

choice calculus (CCC), and refer to the normal form with binary choice as the *binary* choice calculus (2CC) throughout this paper. Syntactically, the binary choice calculus is a normal form of the core choice calculus in which all choices have exactly two alternatives. Consequently, the semantics can be simplified: Instead of picking an index $c(D) \in \mathbb{N}$, configurations c now only have to decide whether to pick the left ($c(D) = \mathit{true}$) or right ($c(D) = \mathit{false}$) alternative of a choice by mapping dimensions D to booleans $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$. As an example,

(2)

encodes the core choice calculus sandwiches from Expression 1 ∈ CCC. The only difference is that the 4-ary choice of Sauce is replaced by nested binary choices. We first decide whether we want Ketchup on our sandwich and in either case subsequently decide whether we want to have Mayo.

3.4 Algebraic Decision Diagrams and Trees (ADT)

Algebraic decision diagrams are graphs that were introduced by Bahar et al. [16] to generalize binary decision diagrams (BDDs) [29, 30] and have been used for algorithmic problems [16, 50] and game theory [2] but also for formalizing variational analyses [33]. Similar to binary decision diagrams [16, 29, 30], algebraic decision diagrams are usually used to solve large instances of computational problems which are usually infeasible and wasteful to represent as trees due to an exponential blowup in size [28, 62, 138]. Most effort in decision diagram research thus is put into identifying shared, equal sub-trees and merging them to obtain a directed, rooted, acyclic graph instead of a tree, hence the name *diagram* and not *tree*. Similar efforts have been made for choice calculus in terms of additional syntax (*let*, *share*, *macro* [136]) to enable sub-tree sharing via references. However, such extensions are (useful) syntactic sugar because the semantics remain unchanged. In this paper, we do semantic comparisons and hence focus on simple trees for reasoning.

Essentially, algebraic decision trees represent series of binary decisions that eventually yield a result. We observe that these binary decisions have the same semantics as choices in choice calculus: For each named decision node, choose the left alternative or the right. Hence, we represent algebraic decision trees with choice-syntax in its grammar given in Table 1. All inner nodes are choices $D\langle l, r\rangle$ and since data can only be stored in leaves, a leaf $\underline{e}$ has to reference an entire variant $\underline{e} \in \mathbb{V}(A)$. Hence, an algebraic decision tree has to enumerate all encoded variants explicitly which causes an exponential blowup because each feature doubles the number of variants. We therefore omit an encoding of our running example from Expression 1 with its $16 = 2 \cdot 2 \cdot 4$ variants here.

3.5 Gruler's Language (GL)

Gruler [64] introduces a formal framework to describe product families and instantiates it for model checking of variational programs [65]. Variability is foremost described in terms of the *variant operator* $e \oplus_n e$ which denotes a binary choice, identified by a number $n \in \mathbb{N}$, analogous to dimensions in the choice calculi.[2] Atoms a may occur only in leaves asset a, two expressions $l, r \in$ GL can be composed via $l \parallel r$, and ntrl is explicit syntax for an empty variant ε. The semantics $[\![e]\!](c)$ of an expression $e \in$ GL resolves all choices as for binary choice calculus.

As an example, we encode the sandwich from Expression 1 and abbreviate asset a as a:

(3)

Since atoms may occur only in leaves, we cannot encode ingredients being *within*  . Crucially, variants described by an expression $e \in$ GL do not describe variants $\underline{e}$ as defined earlier but rather binary trees where only leaves hold atoms, or an empty tree. This means, a variant encodes a list

[2] Gruler's framework also includes a generalization of the variant operator to choices of any arity as in core choice calculus. For simplicity, we stick with to the binary form here.

and not a tree of atoms. Hence, we can only encode which ingredients we need for preparing a sandwich and in which order, but not how to compose them (e.g., put ingredients between slices of ). Formally, there exists in general no bijection between lists and trees and hence associating variants $\underline{e}$ to variants in Gruler's language is ambiguous. Just picking any conversion does not work because there would always be variants that cannot be described. We hence exclude this language from formal comparisons but study it nevertheless because of relevant observations by Gruler [64].

Despite being developed independently of each other, Walkingshaw [136] and Gruler [64] observe the same consequences and laws for choices, including, for example, choice idempotence and distributivity over atoms. Idempotence states that it does not matter which alternative is picked when all alternatives are equal, formulated as $[\![D\langle e, e\rangle]\!] \cong [\![e]\!]$ and $[\![e \oplus_n e]\!] \cong [\![e]\!]$, respectively, where $\cong$ denotes semantic equivalence, as we will define later in Section 4. Distributivity gives rise to powerful transformations to rule out duplicate sub-expressions, formulated as $[\![D\langle a{\prec}e_1{\succ}, a{\prec}e_2{\succ}\rangle]\!] \cong [\![a{\prec}D\langle e_1, e_2\rangle{\succ}]\!]$ and $[\![((\text{asset } a) \parallel e_1) \oplus_n ((\text{asset } a) \parallel e_2)]\!] \cong [\![(\text{asset } a) \parallel (e_1 \oplus_n e_2)]\!]$, respectively.

3.6 Option Calculus (OC)

The languages covered so far, all featured alternative variability in terms of choices. Yet, there is a broad range of works that depict variability in terms of *options* [4, 26, 27, 70, 75, 77, 97]. An option annotates a sub-expression to indicate that it must be either included or excluded from a variant. Options have been used, for example, to decompose legacy applications [75], to extract lost knowledge on variability [97], or for model checking [13, 38, 46–49, 105]. In fact, options are frequently modeled in research and tools that encode variability purely with choices [14, 64, 94, 126, 144]. An option is encoded as a choice with a neutral value ε, just as we did for optional salad in Expression 1. The frequency of this pattern suggests a shared and missed need for options.

Crucially, to the best of our knowledge, there is no rigorous formalization of optional variability. While a definition of syntax is given sometimes [77, 105] (often on examples only), semantics are rarely formalized [105] or even discussed, and most formalisms are tailored to specific use-cases [77] or embedded in specific host languages [47, 105].

Hence, we introduce the *option calculus* as a formal model to depict the nature of options, serving as a common denominator for models based on optional variability. To compare the expressiveness of option-based languages to other languages, option calculus must model variability solely based on options, such that all expressive power stems from options alone. As shown in Table 1, an option calculus expression is built from artifacts $a{\prec}t, \ldots, t{\succ}$ and options $O\langle t\rangle$. An option $O\langle t\rangle$ is identified by a name $O \in \mathbb{F}$ (analogous to dimensions in choice choices) and denotes that the sub-expression t is optional. Each expression is forced to have an artifact at the root in terms of the starting rule e. This is necessary because an option at the top would allow to remove the entire expression, yielding an empty value.

An empty expression is not considered in most models for optional variability, and its meaning is ambiguous (e.g., empty variant vs. non-existence of a variant) and depends on the object language (i.e., the language being configured). If the variability language allows empty expressions in arbitrary places, we might end up with ill-formed variants, such as syntactically incorrect programs. If emptiness is part of the object language instead (i.e., the set of atoms), the empty expression is given meaning by the object language and may only occur in syntactically reasonable places; and the object language determines whether an empty variant exists or has any meaning. To remain general, we must not assume the existence of an empty atom, even though it might exist. When considered explicitly, empty expressions are a source of edge-cases and overhead such as ntrl in Gruler's language [64].

As an example, we encode the sandwich from Section 2 in option calculus:

(4)

Compared to the previous sandwich expressions, all choices have been replaced by options $O\langle e \rangle$, indicated by different braces $\langle\rangle$. For `Salad` and for encoding all combinations of `Ketchup` and `Mayo`, we do not rely on the existence of an "empty" ingredient ε anymore because options natively encode the potential absence of ingredients. Expression 4 is not equivalent to the previous choice calculus expressions 1 and 2 because it denotes more variants ($2^5 = 32 > 16$). Now, sandwiches can have both `Tofu` and `Meat` at the same time or none of them (for configurations c with $c(\texttt{Tofu}) = c(\texttt{Meat}) = \mathit{true}$ or $c(\texttt{Tofu}) = c(\texttt{Meat}) = \mathit{false}$) but having two or no patties should be forbidden according to the specification in our running example in Figure 1. The point is that options cannot encode constraints among the corresponding features which alternatives natively do. We will study this property later in Section 5, including a proof of the incompleteness of option calculus.

The semantics of option calculus are formalized in Table 1 and configure an expression e with a configuration $c : \mathbb{F} \to \mathbb{B}$ by resolving all options with names $O \in \mathbb{F}$ to a variant $\underline{e}$. An option $O\langle e \rangle$ is resolved by either replacing it with the contained expression e upon selection (i.e., $c(O) = \mathit{true}$), or with a temporary placeholder ε upon deselection (i.e., $c(O) = \mathit{false}$). Configuring an artifact $[\![a\langle e_1, \ldots, e_n\rangle]\!](c)$, recursively configures all child expressions and then removes all placeholders ε via an auxiliary function κ with $\kappa([\,]) \coloneqq [\,]$, $\kappa(\varepsilon :: t) \coloneqq \kappa(t)$, and $\kappa(h :: t) \coloneqq h :: \kappa(t), h \neq \varepsilon$.[3] Given that there is always an artifact at the top of an option calculus expression, no placeholders ε remain after configuration such that indeed a variant is produced.

3.7 Feature Structure Trees (FST)

All previous languages model variability by *annotating* a tree, which simultaneously contains all variants. An orthogonal paradigm is *compositional* variability in which features are divided into distinct modules and composed to a variant, as done in aspect-oriented [83, 84], delta-oriented [98, 117], or feature-oriented programming [9], or plug-in-frameworks [71] and mixins [124].

Apel et al. [11] present an algebra that abstracts compositional static variability, and which covers different implementations of feature-oriented [9, 10, 12, 19], aspect-oriented [8, 82, 125], or other compositional languages [22, 101]. This algebra basically assumes the existence of a set of so called *introductions* I (e.g., features or modules) which can be composed or modified via various binary operators while obeying algebraic laws. For example, the *introduction sum* $\oplus : I \times I \to I$, which composes two introductions, must obey *distant idempotence* $a \oplus b \oplus a = a \oplus b$ which means that adding an introduction, which has already been added has no effect.[4]

In this paper, we focus on one compositional language that is an instance of the algebra: feature structure trees [9, 11]. In Table 1, we show simplified syntax and semantics of feature structure trees. Originally, the syntax is defined in natural language [9] or as sets of paths to represent trees [11], which we simplify to a small grammar. Basically, an expression is a list of features $a \blacktriangleleft f, \ldots, f \blacktriangleright$ which can be inserted as sub-expressions of a common atom a (e.g., the root directory of a software project). Each feature $F : \underline{e}, \ldots, \underline{e}$ has a name $F \in \mathbb{F}$ and a range of trees, represented as plain variants, which are introduced when the feature is selected. For example, the feature `Weight : package graph≺class Edge≺int weight≻, class Node≺int weight≻≻` adds a field `int weight` to two Java classes in a graph library. To derive a variant, the semantics compose the trees of all selected features, via a binary operator $\oplus$:

[3] The function κ is also known as `catMaybes` in the standard libraries of Haskell and Agda.

[4] Any non-commutative introduction sum $\oplus$ has to choose whether left introductions dominate right introductions or vice versa, giving rise to two symmetric distant idempotence laws. We choose the left-dominant version here (see above) because it seems more intuitive to us, whereas Apel et al. [11] present the right-dominant formulation $a \oplus b \oplus a = b \oplus a$.

$$\oplus : \underline{e}^n \to \underline{e}^m \to \underline{e}^o \qquad \odot : \underline{e}^n \to \underline{e} \to \underline{e}^m$$

$$l \oplus [\,] := l \qquad [\,] \odot b\text{-}\prec c_b \succ\text{-} := [b\text{-}\prec c_b \succ\text{-}]$$

$$l \oplus (h :: t) := (l \odot h) \oplus t \qquad \left(a\text{-}\prec c_a \succ\text{-} :: t\right) \odot b\text{-}\prec c_b \succ\text{-} := \begin{cases} a\text{-}\prec c_a \oplus c_b \succ\text{-} :: t, & a = b, \\ a\text{-}\prec c_a \succ\text{-} :: \left(t \odot b\text{-}\prec c_b \succ\text{-}\right), & a \neq b \end{cases}$$

Composition $l \oplus r$ composes all trees in a list $r \in \underline{e}^m$ onto the trees in a list $l \in \underline{e}^n$ sequentially from left to right via the operator $\odot$.[5] The operator $\odot$ composes a single tree $b\text{-}\prec c_b \succ\text{-}$, where c_b is a list of children, into a list of trees by sequentially inspecting each tree in the list: When a tree $a\text{-}\prec c_a \succ\text{-}$ is found with the same atom $a = b$, then both trees are composed recursively. When no matching tree is found, the tree to compose $b\text{-}\prec c_b \succ\text{-}$ is appended to the list as a new implementation. This case distinction allows to modify (case $a = b$) or add to (case $a \neq b$) an existing implementation.

As an example, we can compose a sandwich, where each feature's list holds one ingredient.

(5)

Given that ingredients are flat trees and that no ingredient occurs more than once here, composition $\odot$ will always append but never merge ingredients (case $a \neq b$). For example, $[\![\text{Expression 5}]\!](c) =$  if $c(\texttt{Salad}) = c(\texttt{Cheese}) = \mathit{true}$ but false otherwise.

3.8 Other Languages & Conclusion

There are many other models for variability, but most of them are tied to specific use cases or (programming) languages, including models with binary [15, 102, 107] or n-ary choices [140–142], or more complex choice semantics [121]. In fact, the choice calculus itself has more dialects [69, 136]. Some languages mix options and alternatives [24, 27, 80, 139], which we will discuss later.

The variability languages we covered so far are generic in the sense that they do not make assumptions on the semantics nor syntax of the language being configured (i.e., the object language) except that it's syntax should be tree-like. Other languages, such as languages used for variability-aware syntax- [78] and type-checking [79, 94], or control flow analyses [15, 102, 107, 140–142] are aware of the object language but at the cost of being tied to a specific set of object languages such as transition systems [49, 133] or programming languages [47, 76, 121, 140]. While specializing for specific object languages enables for optimizations and more effective reasoning in that object language, our goal is to study how to express variability by itself. We hence must study only generic languages. In some sense, the generic languages presented here are also representative for non-generic languages because non-generic languages extend an object language's grammar with additional rules for variability, which are the same constructs used in generic languages (e.g., flavors of choices).

In the next section, we develop a formal framework to describe and compare static variability languages. In Section 5, we then compare the languages presented here, but our framework will be general enough to also cover other languages and dialects as well.

4 A Formal Framework for Language Comparisons

With a zoo of informal and formal variability languages employed in various research efforts, we now turn to developing a formal framework to compare and study languages.

[5] Functional programmers might recognize $\oplus$ as a (left) fold on lists.

4.1 Semantic Domain of Variability Languages

To compare the semantic expressiveness of formal variability languages, such as those illustrated in Section 3, we must compare their ability to describe elements of their semantic domain. Research on variational systems and software product lines is typically based on the shared intuition that a variational expression (i.e., a product line) specifies a set of variants in a target language, such as programming languages or sandwiches. Each variant is commonly identified by a configuration (cf. Section 2 and 3.1), which means that a variant can be automatically *generated* or otherwise *retrieved* from the product line by means of evaluating the product line against a configuration. Besides work specifically focused on the choice calculus (cf. Section 3.3), examples for this intuition are variability-aware analysis [25, 46, 47, 65, 141, 144], other formal frameworks [33, 49] or surveys [134] for product line analyses, variation control systems [95, 96, 120, 126], managed clone-and-own development [81, 99, 112, 116, 118], as well as a popular software product lines text book [7]. In Section 3.1, we observed that variants are modeled by variability languages as trees of atomic values, and provided definitions for atoms and variants in Definition 3.1, respectively.

However, sets of variants are exponential in the number of configuration options. Computing all configurations and their respective variants does not scale in practice and requires to generate all possible configurations combinatorically. We can avoid explicitly computing sets of configurations and the complexity introduced by the respective combinatorics though. The key idea is to encode a set of variants as a function $f : C \to \mathbb{V}(A)$ that *selects a set* of variants $Im(f) \subset \mathbb{V}(A)$ from the set of all variants $\mathbb{V}(A)$ over atoms A in terms of an index set C (e.g., configurations). This formulation separates the concerns of (1) describing how to generate a variant, and (2) quantifying all variants. In fact, we can compare semantics solely in terms of (1) variant generation functions f, without (2) ever having to compute the actual set of variants which is nevertheless possible as we will cover later. In our experience, this formulation allows for more elegant and concise argument and proof, and explains our functional style for semantics in Section 3.

Definition 4.1 (Indexed Set). An indexed set is a function $A : I \to S$, which associates each index $i \in I$ with an element $A(i) \in S$.

The key idea in Definition 4.1 is that we can model a subset of a set S by *pointing* at those elements we want to have in the subset. This pointing is done in terms of a function that maps indices or keys $i \in I$ to the elements in S. We are not the first to have this idea: identifying sets of objects by pointing is also known in the context of *elements*, *points*, or *subobjects* in category theory [17] or *object classifiers* in homotopy type theory [135].

Example 4.2. A function $f : \mathbb{B} \to \mathbb{N}$ is an indexed set of at most two natural numbers. When $f(false) = 1$ and $f(true) = 2$ then f denotes $\{1, 2\}$. As another example, the function $even : \mathbb{N} \to \mathbb{N}$ with $even(n) = 2 \cdot n$ is an indexed set that denotes the subset of all even natural numbers.

When an indexed set is not injective, then it denotes a multiset because there exists at least one element that is pointed at twice. This aligns with the semantics of variability languages because the same variant might be associated with different configurations. For example, in the expression $D\langle e, e\rangle \in \mathtt{2CC}$ a configuration which picks the left alternative for D remains equivalent if it instead would pick the right alternative because $[\![D\langle e, e\rangle]\!](c) = [\![e]\!](c)$ for all configurations c (cf. Section 3.5). When an indexed set is not surjective it denotes a proper subset of S because there are elements in S that are not being pointed at, such as in $even$.

Indexed sets can be compared based on the usual operator for set inclusion:

Definition 4.3 (Indexed Set Inclusion $\sqsubseteq$, Subset $\sqsubseteq$, and Equivalence $\cong$). An element $s \in S$ is an element of an indexed set $A : I \to S$ if there exists an index $i \in I$ that points to s. Formally, we

write $s \sqsubseteq A$ iff $\exists i \in I : A(i) =_S s$ for an equivalence relation $=_S$ over S. An indexed set $A : I \to S$ is a subset of an indexed set $B : I \to S$ if B points to all elements A points to. Formally, we write $A \sqsubseteq B$ iff $\forall i \in I : A(i) \sqsubseteq B$. We consider two indexed sets equivalent and write $A \cong B$ iff $A \sqsubseteq B$ and $B \sqsubseteq A$.

Example 4.4. Let $A : \mathbb{B} \to \mathbb{N}$ and $B : \{\diamond, \heartsuit, \spadesuit, \clubsuit\} \to \mathbb{N}$ be two indexed sets with $A(\mathit{false}) = A(\mathit{true}) = 3$ and $B(\diamond) = 1$, $B(\heartsuit) = 2$, $B(\spadesuit) = 3$, and $B(\clubsuit) = 4$. Then, $A \sqsubseteq B$ because $A(\mathit{false}) \sqsubseteq B$ and $A(\mathit{true}) \sqsubseteq B$ because $A(\mathit{false}) = B(\spadesuit)$ and $A(\mathit{true}) = B(\spadesuit)$.

Corollary 4.5. *$\sqsubseteq$ is a partial order and $\cong$ is an equivalence relation.*

In this paper, we cover variability languages that describe *finite* sets of variants because that is what formal variability languages typically describe (cf. Section 3). Infinite variant sets occur in practice when configuration options $\mathbb{F}$ can have unbounded domains such as numbers or strings. However, such variability is typically implemented within a programming language and cannot be expressed with annotations, which would require infinitely many trees below annotations, or compositions, which would require an infinite amount of components or modules. While supported by our theory in principle, infinite variant spaces are out of scope here.

Definition 4.6 (Finite Indexed Set). An indexed set $A : I \to S$ is finite iff the set I is finite.

When the set of indices I is finite, the indexed set A can only pick a finite amount of elements from S and hence the indexed set A is finite. For upcoming definitions and proofs we use $\mathbb{N}_n \coloneqq \{1, \ldots, n\}, n \in \mathbb{N}$ as a canonical finite set that always contains exactly n elements.[6] To prove that a set of indices I is finite, we enumerate it with a surjective function $\mathit{enum} : \mathbb{N}_n \to I$.

Example 4.7. The sets A and B from Example 4.4 are finite because their index sets are finite. The indexed set even from Example 4.2 is infinite because it ranges over all natural numbers.

Moreover, expressions in variability languages should denote *non-empty* subsets of variants. The reason is that denoting an empty set of variants would require the existence of at least one *empty* expression, which does not exist in common formal variability languages, has unclear meaning and soundness issues (cf. Section 3.6). An indexed set $A : I \to S$ is non-empty if it points to at least one element $s \in S$, which in turn means that there must be at least one index $i \in I$.

Definition 4.8 (Non-empty Indexed Set). An indexed set $A : I \to S$ is non-empty iff $I \neq \emptyset$.

Finally, we define the semantic domain of variability languages in terms of finite and non-empty indexed sets of variants which we call variant generators.

Definition 4.9 (Variant Generator). A variant generator is a finite and non-empty indexed set $I \to \mathbb{V}(A)$ of variants over atoms A with an otherwise arbitrary index set I.

Definition 4.10 (Semantic Domain). The semantic domain $\mathbb{S}(A)$ of variability languages is the set of all variant generators over atoms A.

Having defined the non-variational elements of our framework — atoms and variants — and the semantic domain in terms of variant generators, we can now turn to implementing variability.

4.2 Variability Languages

An expression in a variability language associates variants with configurations (cf. Section 2). Configurations encode the necessary information to configure a variational expression to a variant. Different variability languages may require configurations of different forms though.

[6] $\mathbb{N}_n$ is also known as `Fin n` in proof assistants.

Definition 4.11 (Configuration Language). A configuration language is a set C.

We have seen examples for different configuration languages in Section 3. Our framework does not impose any assumptions on how a configuration is structured and thus, we can depict a configuration language as a mere set. We can now define an abstract notion of variability languages.

Definition 4.12 (Variability Language). A variability language L is a set of expressions $L(A)$ that is parameterized in an atom set A.

We define the denotational semantics $[\![e]\!]$ of an expression $e \in L(A)$ of a variability language as an indexed set where indices are configurations:

Definition 4.13 (Denotational Semantics of Variability Languages). The denotational semantics of a variability language L with a configuration language C is a function $[\![.]\!] : L(A) \to C \to \mathbb{V}(A)$, which configures an expression $e \in L(A)$ with a configuration $c \in C$ to a variant $[\![e]\!](c) \in \mathbb{V}(A)$, for any atom set A.

Example 4.14. All languages illustrated in Section 3 have denotational semantics according to this definition. For instance, the semantics of binary choice calculus is a function $[\![.]\!] : \mathtt{2CC}(A) \to C \to \mathbb{V}(A)$, where configurations $C : \mathbb{F} \to \mathbb{B}$ map dimensions to Boolean values. The semantics $[\![e]\!]$ of an expression e then is a function $C \to \mathbb{V}(A)$ and hence an indexed set. For example, $[\![D\langle 1, 2\rangle]\!]$ is an indexed set of two variants 1 and 2 because $[\![D\langle 1, 2\rangle]\!](\lambda x.\ \mathit{true}) = 1$ and $[\![D\langle 1, 2\rangle]\!](\lambda x.\ \mathit{false}) = 2$.

While we defined the semantic domain of variability languages in terms of variant generators (i.e., non-empty, finite indexed sets of variants) earlier, we do not constrain the denotational semantics of a variability language to variant generators. This allows to represent unsound languages within the framework as well to analyze whether a language is indeed sound or not, and to nevertheless relate it to other languages. We now introduce completeness and soundness to check if a variability language denotes variant generators.

4.3 Completeness, Soundness, and Expressiveness

A key property of a language's denotational semantics is *completeness*. A language is complete if it is capable of expressing each element in its semantic domain. In this paper, we consider the semantic domain of variability languages to be variant generators (Definition 4.10). We thus consider a variability language to be *complete* iff it can express any variant generator $V : \mathbb{N}_n \to \mathbb{V}(A)$ for any atom set A , where we use $\mathbb{N}_n$ as a canonical non-empty and finite indexed set, independent from any particular configuration knowledge. In a complete language, for example, we can describe any set of products demanded by customers, or in terms of our running example, a recipe book for any set of sandwiches.

Definition 4.15 (Completeness). A variability language is complete iff it can express any variant generator. Formally, we write $\mathrm{Complete}(L)$ iff $\forall A, n \in \mathbb{N}, (V : \mathbb{N}_n \to \mathbb{V}(A))\ \exists e \in L(A) :\ [\![e]\!] \cong V$.

The converse property is *soundness*. Soundness ensures that each expression indeed denotes a variant generator, and hence has meaning in the semantic domain. If a variability language is not sound, some of its expressions describe things other than variant generators, and hence systems that are ill-formed or meaningless to the variational domain for a given atom set. In terms of our running example, a language for cookbooks for sandwiches would be unsound if it for example allows to describe also things that are not cookbooks at all, or that are cookbooks that also include recipes for making spaghetti or a space rocket.

Definition 4.16 (Soundness). A variability language is sound iff it denotes only variant generators. Formally, we write $\mathrm{Sound}(L)$ iff $\forall A, e \in L(A)\ \exists n \in \mathbb{N}, (V : \mathbb{N}_n \to \mathbb{V}(A)) :\ [\![e]\!] \cong V$.

The major goal of this paper is to determine *whether* and *how* research based on one variability language can be transferred to other variability languages. Therefore, we need a way to relate languages based on their semantics, commonly referred to as expressiveness [41, 43, 56]: Can a language describe the semantics of another language? With completeness, we already have a measure for expressiveness in absolute terms (i.e., whether a language can describe *all* variant generators). For comparing languages, we are interested in a *relative* measure that tells us whether a language is less or more expressive than another language.

Definition 4.17 (Expressiveness Relations $\succeq$, $\equiv$, $\succ$). Let L, M be two variability languages with semantics $[\![.]\!]_L$, $[\![.]\!]_M$, respectively. L is at least as expressive as M if any indexed set $[\![m]\!]_M$ that can be described by an expression $m \in M$, can also be described by an expression $l \in L$. Formally, we write $L \succeq M$ iff $\forall A, m \in M(A)\ \exists l \in L(A) : [\![l]\!]_L \cong [\![m]\!]_M$. Two variability languages are equally expressive $L \equiv M$ iff they are at least as expressive as each other, $L \succeq M$ and $M \succeq L$. A variability language is more expressive than another variability language $L \succ M$ iff L is at least as expressive as M but not vice versa, $L \succeq M$ and $M \not\succeq L$.

The intuition behind this notion of expressiveness is that a language L can express "everything" another language M can express if $L \succeq M$. Following the naming convention for comparing expressions within choice calculus [53], we refer to two expressions $l \in L, m \in M$ that describe the same variants but with potentially different configurations, $[\![l]\!]_L \cong [\![m]\!]_M$, as *variant-equivalent*.

Corollary 4.18. *$\succeq$ is a partial order and $\equiv$ is an equivalence relation.*

4.4 How to Prove Completeness, Soundness, and Expressiveness?

Having introduced the core definitions of our framework, including completeness, soundness, and expressiveness, we must demonstrate how to prove these properties for a given language. We now explain how to construct direct proofs, and enrich our framework with theorems that leverage the relationship between completeness, soundness, and expressiveness to conclude proofs for free.

Completeness by Differencing. We can prove completeness by implementing an encoding algorithm and showing its correctness. Given a variant generator $V : \mathbb{N}_n \to \mathbb{V}(A)$, this algorithm must construct an expression e that encodes exactly the variants in that variant generator (i.e., $[\![e]\!] \cong V$). Encoding a set of trees V into a single variational expression e that describes their similarities and differences is also known as tree differencing [54, 60] and part of migrating cloned variants or forks into a software product line [57–59, 81, 92, 115, 146].

Soundness by Enumeration. To prove soundness of a language, we must show that every expression denotes a variant generator. Therefore, we can compute the variant generator $V : \mathbb{N}_n \to \mathbb{V}(A)$ of a given expression e and prove correctness of this computation (i.e., $[\![e]\!] \cong V$). The semantics $[\![e]\!]$ of an expression e is an indexed set $C \to \mathbb{V}(A)$ from configurations to variants. We can convert $[\![e]\!]$ to V by enumerating all configurations in terms of a surjective function $enum : \mathbb{N}_n \to C$ as discussed for Definition 4.6, such that $V = [\![e]\!] \circ enum$. Proving $enum$ surjective, proves that C is finite and non-empty. For some languages there might be infinitely many configurations though, as for example with choice calculus configurations $\mathbb{F} \to \mathbb{N}$ (cf. Table 1). Yet, configurations still describe only a finite amount of variants because each expression is finite. For example, all configurations $c_i(x) = i$ with $1 < i \in \mathbb{N}$ produce the same variant  $= [\![D\langle$   $\rangle]\!](c_i)$. Hence, for an expression e, we can consider two configurations c_1, c_2 equivalent iff they produce the same variant $[\![e]\!](c_1) = [\![e]\!](c_2)$, as also observed by Gruler [64, p. 66]. We then must prove surjectivity of $enum$ with respect to configuration equivalence.

$$\frac{\mathrm{Complete}(M) \qquad L \succeq M}{\mathrm{Complete}(L)}\ \text{Th. 4.22} \qquad \frac{\mathrm{Sound}(L) \qquad L \succeq M}{\mathrm{Sound}(M)}\ \text{Th. 4.23} \qquad \frac{\mathrm{Complete}(L) \qquad \mathrm{Sound}(M)}{L \succeq M}\ \text{Th. 4.24}$$

Fig. 2. Theorems relating completeness, soundness, and expressiveness.

Expressiveness by Compilation. To prove $L \succeq M$, we can compile M to L to show that every expression in M has a corresponding expression in L; and hence L can express "everything" M can express. Compilation entails translating expressions and proving correctness: A translated expression must denote the same variant generator (i.e., be variant-equivalent). Given a translation $\mathrm{t} : M \to L$ this means proving $\forall m \in M : [\![m]\!]_M \cong [\![\mathrm{t}(m)]\!]_L$ which reduces to $[\![m]\!]_M \sqsubseteq [\![\mathrm{t}(m)]\!]_L$ and $[\![\mathrm{t}(m)]\!]_L \sqsubseteq [\![m]\!]_M$. These further reduce to $\forall c_M \in C_M\ \exists c_L \in C_L : [\![m]\!]_M(c_M) = [\![\mathrm{t}(m)]\!]_L(c_L)$, and $\forall c_L \in C_L\ \exists c_M \in C_M : [\![\mathrm{t}(m)]\!]_L(c_L) = [\![m]\!]_M(c_M)$. Hence, we must also translate the respective configuration languages C_L, C_M. When a translation t alters annotations in m (e.g., splitting a choice as in Expression 2, thereby introducing new dimensions), translating configurations correctly also depends on the input expression.

Definition 4.19 (Compiler). Let M, L be variability languages. Let C_M and C_L be the corresponding configuration languages. A compiler is a 3-tuple $(\mathrm{t}, \mathrm{conf}, \mathrm{fnoc}) \in M \mathrel{-\!\!\triangleright} L$, where $M \mathrel{-\!\!\triangleright} L = (M \to L) \times (M \to C_M \to C_L) \times (M \to C_L \to C_M)$ is the set of all compilers from M to L, t is a translation for expressions and conf and fnoc are translations of configurations.

Definition 4.20 (Compiler Correctness). Let M, L be variability languages with configuration languages C_M, C_L and semantics $[\![.]\!]_M, [\![.]\!]_L$. A compiler $(\mathrm{t}, \mathrm{conf}, \mathrm{fnoc}) \in M \mathrel{-\!\!\triangleright} L$ is correct if $\forall m \in M$, $c_M \in C_M, c_L \in C_L$: $[\![m]\!]_M(c_M) = [\![\mathrm{t}(m)]\!]_L(\mathrm{conf}(m, c_M))$ and $[\![\mathrm{t}(m)]\!]_L(c_L) = [\![m]\!]_M(\mathrm{fnoc}(m, c_L))$.

Theorem 4.21 (Expressiveness by Compilation). *Let L, M be two variability languages. $L \succeq M$ if there exists a correct compiler $M \mathrel{-\!\!\triangleright} L$.*

Proof. Let A be an atom set. To prove $L \succeq M$, we must show $\forall m \in M(A)\ \exists l \in L(A) : [\![l]\!]_L \cong [\![m]\!]_M$, which reduces to $\forall c_L \in C_L : \exists c_M \in C_M : [\![l]\!]_L(c_L) = [\![m]\!]_M(c_M)$, and $\forall c_M \in C_M : \exists c_L \in C_L : [\![m]\!]_M(c_M) = [\![l]\!]_L(c_L)$. Both propositions hold by definition of compiler correctness, where l is given by $\mathrm{t}(m)$ and the configurations exist by $\mathrm{conf}(m, c_M)$ and $\mathrm{fnoc}(m, c_L)$. □

Proofs for Free. Figure 2 depicts how completeness, soundness, and expressiveness interact, giving rise to theorems to conclude proofs for free. As usual, completeness and soundness are converse to each other causing them to have dual relationships to expressiveness. When a language L is at least as expressive as a complete language M, then L is also complete because L is able to describe any set of variants which the complete language M can encode (Theorem 4.22).

Proof. Let A be an atom set. Let V be a variant generator. We have to show that V can be described by an expression $l \in L(A)$ such that $[\![l]\!]_L \cong V$. By completeness of M, there exists an expression $m \in M(A)$ with $[\![m]\!]_M \cong V$. By expressiveness $L \succeq M$, there exists an expression $l \in L(A)$ with $[\![l]\!]_L \cong [\![m]\!]_M$. By transitivity of $\cong$, we conclude $[\![l]\!]_L \cong V$. □

Conversely, when a sound language L is least as expressive as another language M, then also that other language M is sound because it describes at most the sets of variants described by the sound language L, which in turn are variant generators (Theorem 4.23).

Proof. Let A be an atom set. Let $m \in M(A)$ be an expression in M. We have to show that there exists a variant generator V of variants such that $[\![m]\!]_M \cong V$. By expressiveness $L \succeq M$, there is

an expression $l \in L(A)$ with $[\![l]\!]_L \cong [\![m]\!]_M$. By soundness of L, the expression l denotes a variant generator V with $[\![l]\!]_L \cong V$. By transitivity and symmetry of $\cong$, we can conclude that $[\![m]\!]_M \cong V$. □

A key observation is that a complete language is at least as expressive as any sound language, denoted by Theorem 4.24. Intuitively, a language L that can encode any variant generator, can express any expression of another language M if these expressions indeed describe variant generators.

Proof. Let A be an atom set. To prove that $L \succeq M$, we have to show that for all expressions $m \in M(A)$, there exists an expression $l \in L(A)$ with $[\![l]\!]_L \cong [\![m]\!]_M$. Let $m \in M(A)$. Since M is sound, there exists a variant generator V with $[\![m]\!]_M \cong V$. By completeness of L, there exists an expression $l \in L(A)$ with $[\![l]\!]_L \cong V$. By transitivity and symmetry of $\cong$, we conclude $[\![l]\!]_L \cong [\![m]\!]_M$. □

This interplay between completeness, soundness, and expressiveness, is in fact not specific to variability languages but may emerge in any system with these three properties. There are even more interesting theorems, for example to conclude incompleteness or unsoundness, or to conclude that no sound language can be more expressive than a complete language. We omit these theorems for brevity here but formalize and prove them in our Agda implementation [1].

4.5 Formalization, Tool Support, and Conclusion

We formalize our framework and all its proofs as an open source Agda library [1]. Thereby our framework addresses both theoretical and practical concerns. For language designers, our framework enables basic sanity checks, which despite being basic, have never been studied nor implemented for variability languages before. For practitioners, our framework provides proven-to-be correct compilers and differencing algorithms, originating from constructive proofs for the above properties, implemented in Agda. For researchers, our framework provides proof strategies to derive insightful theorems on the expressiveness, completeness, and soundness of variability languages.

5 Charting the Language Space

We now begin the journey to chart the landscape of formal variability languages to bridge the gaps between existing research efforts. We contribute (1) a semantic comparison of common variability languages, which also serves as (2) a case study to show the feasibility and usability of our framework, and (3) a web of compilers and correctness proofs, formalized in Agda [1].

Figure 3 gives an overview of the landscape we discovered and organized in a systematic way. In this graph, a node represents a variability language and an edge (L, M) denotes the existence of a correct compiler $L \rightarrowtriangle M$ (cf. Definition 4.19 and 4.20). An edge is crossed out $L \not\rightarrowtriangle M$ if there cannot exist a correct compiler. Loops denote intra-language compilers $L \rightarrowtriangle L$. An edge has a filled tip $L \blacktriangleright M$ if there exists a compiler $L \rightarrowtriangle M$ and $L \subseteq M$ by definition, which renders constructing a compiler trivial.[7] In the following, we omit explicitly stating corollaries corresponding to such edges $L \blacktriangleright M$. We formalize all compilers, theorems, corollaries, and their proofs in our Agda library Vatras [1]. The library's documentation contains a detailed mapping of the theorems and proofs in this paper to their respective Agda representation.

Finding compilers *and* proving their correctness is a means to prove completeness, soundness, or expressiveness (cf. Section 4.4). For each edge $L \rightarrowtriangle M$ in Figure 3, we conclude $M \succeq L$ by Theorem 4.21. Transitively, we can conclude completeness by Theorem 4.22, if a language we

[7] When described in type theory, as we do in our Agda library, a language is a type and an expression is an instance of that type, and hence every expression belongs to exactly one language. Hence, relating languages always requires explicit translation functions, even if conceptually, the terms of one language are a subset of the other language. In our library, we provide such conversions and correctness proofs, but we leave them out in this chapter for clarity and brevity. When necessary, we point out such a conversion with the identity function *id* (e.g., in Figure 3).

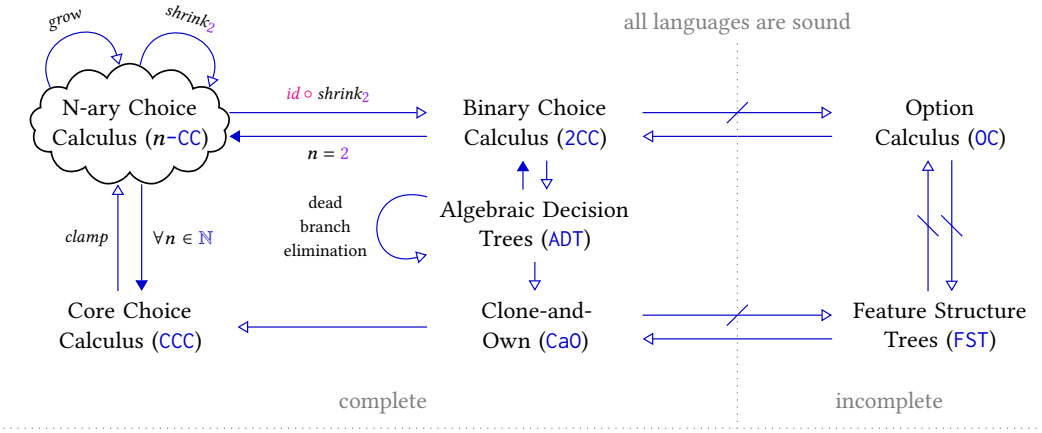


Fig. 3. Compilation map of common variability languages.

compiled is complete. This means that completeness "flows" along edges in Figure 3 and, thus, proving a language complete, proves all reachable languages complete. Similarly, any language from which we can reach a sound language is also sound by Theorem 4.23.

To relate each language to each other language, we try to construct circles of compilers instead of doing pairwise comparisons. Finding a circle proves all contained languages equally expressive: Any language L_i within a circle $L_1 \twoheadrightarrow L_2 \twoheadrightarrow \ldots \twoheadrightarrow L_n \twoheadrightarrow L_1$ can be translated to any other language L_j because compilation is transitive. Hence, circles enable pairwise comparisons with only n instead of $n(n-1)$ compilers. Key here is to find circles that are the easiest to implement and prove correct.

5.1 Choice-based Languages

We start constructing a circle of compilers with the choice-based languages core choice calculus (CCC), binary choice calculus (2CC), and algebraic decision trees (ADT). To relate the choice calculi, we introduce the language family of n-ary choice calculus (n-CC): a language similar to CCC but every choice has *exactly* $n \geq 2$ alternatives. The semantics for n-CC is the same function as for core choice calculus shown in Table 1. In Figure 3, n-CC is represented by the cloud to symbolize that n-CC is an infinite family of languages parameterized in a natural number. Edges going into the cloud translate to a single dialect with a specific n, and edges going out of the cloud translate from all dialects to the respective language.

Theorem 5.1. $\forall n \in \mathbb{N}, n \geq 2 : n\text{-CC} \succeq \text{CCC}$.

Proof Sketch by Compilation. Let $e \in \text{CCC}, n \in \mathbb{N}, n \geq 2$. We must show that there exists $e' \in n\text{-CC}$ with $[\![e]\!]_{\text{CCC}} \cong [\![e']\!]_{n\text{-CC}}$. We construct and compose three reusable compilers.

(1) clamp $\in \big((e \in \text{CCC}) \twoheadrightarrow \lceil e \rceil\text{-CC}\big)$. Because choices in $e \in \text{CCC}$ may have arbitrary and different numbers of alternatives, we first have to unify choices to a fixed arity $m = \lceil e \rceil$, where $\lceil e \rceil$ computes the maximum number of alternatives of choices in e. We turn each choice $D\langle x_1, \ldots, x_k \rangle, k < \lceil e \rceil$ with fewer alternatives into a choice with exactly $\lceil e \rceil$ alternatives by replicating the last alternative x_k, $\lceil e \rceil - o$ times, in terms of a function *fill*, giving us $e'' = \mathit{fill}(e, \lceil e \rceil) \in \lceil e \rceil\text{-CC}$. To prove the translation correct, we must show $[\![e]\!] \cong [\![e'']\!]$ (cf. Definition 4.19). Fortunately, configurations remain constant and do not have to be translated here. The idea of the proof is to check whether a given configuration $c : \mathbb{F} \to \mathbb{N}$, for a given choice $D\langle x_1, \ldots, x_k \rangle$ in $e \in \text{CCC}$, chooses an alternative in bounds $c(D) \leq k$ or above $k < c(D) \leq \lceil e \rceil$. For $c(D) \leq k$, both expressions yield the same

alternatives by definition. For $k < c(D) \leq \lceil e \rceil$, $c(D)$ always picks x_k in both expressions because $[\![D\langle x_1, \ldots, x_k\rangle]\!](c) = [\![x_{min(c(D),k)}]\!](c)$ clamps the value of $c(D)$ to k by definition, and within e' the choice contains only duplicates of x_k at the respective spots. The actual formal proof of correctness in our Agda library is more complex as it involves a custom type system for CCC expressions to prove them having at most $\lceil e \rceil$ alternatives, proofs that configurations for $\lceil e \rceil$-CC remain in bounds, and various lemmas for *fill*. This leaves us with a compiler $clamp \in \big((e \in \text{CCC}) \rightarrowtriangle \lceil e \rceil\text{-CC}\big)$, where the output language is selected based on the input expression e. To avoid this dependency, we show that we may freely change the arity n of any n-CC expression.

(2) shrink$_2 \in m\text{-CC} \rightarrowtriangle 2\text{-CC}$. To remove the dependency of the output language on the input expression, we show that we can reduce any arity m to any smaller value $k < m$. For brevity, and to prepare for translating CCC to 2CC, we simply specialize $k = 2$. The idea of the translation is to nest surplus alternatives in new choices, similar how a long if-elif*-else chain can be converted to a nested if-else statement in imperative programming, or how lists may be represented as recursive pairs $h :: t$ of head element h and tail list t. This nesting requires to introduce new choices and hence new dimensions, which we do by associating each dimension in the input expression with an additional index $i \in \mathbb{N}$. For instance, we can translate $D\langle$  $\rangle$ to $D_1\langle$ $, D_2\langle$  $\rangle\rangle$. To prove this translation correct, we have to translate configurations $\mathbb{F} \to \mathbb{N}_n$ to configurations $\mathbb{F} \to \mathbb{B}$ and back, which requires mapping selections of dimensions to boolean decisions on indexed dimensions, and vice versa. For instance, in the above example, a configuration $c : \mathbb{F} \to \mathbb{N}_n$ with $c(D) = 2$ must translate to a configuration c' with $c'(D_1) = \mathit{false}$ and $c'(D_2) = \mathit{true}$.

(3) grow $\in k\text{-CC} \rightarrowtriangle m\text{-CC}, \forall m > k$. By a similar argument as for *clamp*, we can increase the arity of every choice in an n-CC term by duplicating the last alternative $m - k$ times.

Conclusion. By composing the above compilers and their correctness proofs, we obtain a compiler CCC $\rightarrowtriangle n$-CC for our freely chosen $n \in \mathbb{N}, n \geq 2$ (remember, that $\cong$ is transitive by Corollary 4.5). □

Example 5.2. As an example, we translate a variational sandwich recipe $e_1 =$  $\prec$P$\langle$  $,$  $\rangle$, S$\langle$  $,$  $\rangle\succ \in$ CCC to $\lceil e_1 \rceil$-CC and then 2CC. We find $\lceil e_1 \rceil = 3$ because P$\langle\ldots\rangle$ has 2 alternatives but S$\langle\ldots\rangle$ has 3. Via *clamp* we fill all choices to have 3 alternatives, we get $e_2 = \mathit{fill}(e_1, 3) =$  $\prec$P$\langle$  $,$  $,$  $\rangle$, S$\langle$ $,$ $\rangle\succ \in$ 3-CC. Via *shrink*$_2$, we then nest choices recursively to become binary, giving us $e_3 =$  $\prec$P$_0\langle$  $,$ P$_1\langle$  $,$  $\rangle\rangle$, S$_0\langle$ $,$ S$_1\langle$ $\rangle\rangle\succ \in$ 2CC. The inner choice P$_1\langle\ldots\rangle$ with all the same alternatives could be simplified by choice idempotence (cf., Section 3.5), for example by constructing another intra-language compiler to eliminate redundancy.

Next up, are ADT, which were developed independently from the other languages and without focus on variability analyses. We observe that in fact, ADT is a *normal form* of 2CC, where all artifacts are leaves (i.e., they have no sub-expressions).

Theorem 5.3. ADT $\succeq$ 2CC.

Proof Sketch by Compilation. To translate 2CC to ADT, we must ensure that there are no choices below artifacts. Choices must be factored upwards, which is possible by duplicating atoms and known as choice distribution [64] or factoring [136] (cf. Section 3.5): $a \prec D\langle l, r\rangle, e_2, \ldots, e_n \succ$ can be factored to $D\langle a \prec l, e_2, \ldots, e_n \succ, a \prec r, e_2, \ldots, e_n \succ\rangle$. The translation recursively factors all choices upwards, and then replaces the largest subtrees that solely consist of artifacts by leaf nodes that reference the subtree as a variant $\underline{e}$. The correctness proof works by induction. □

Factoring introduces an exponential blow-up, because all neighboring sub-expressions are duplicated until eventually all variants of e are enumerated in leaves and no atoms are shared. In fact, it is this blow-up that motivates artifact nodes *within* expressions in choice calculi to share

similar subtrees [136], and motivates software product-line analyses because analyzing or even enumerating each variant individually is infeasible in practice because of this blow-up [33, 134].

Example 5.4. We continue our previous example $e_3 \in \texttt{2CC}$ but simplify the inner idempotent choice of $\texttt{P}_1$ and start with $e_4 = \langle \texttt{P}_\emptyset\langle \langle \rangle, \langle \rangle \rangle, \texttt{S}_\emptyset\langle \langle \rangle, \texttt{S}_1\langle \langle \rangle, \langle \rangle \rangle \rangle \rangle \in \texttt{2CC}$. Only $\langle \rangle$ is an atom above choices. When factoring $\langle \rangle$ with the first choice for the patty $\texttt{P}_\emptyset$, we get a choice at the top which contains two expressions in which the patty is fixed: $e_5 = \texttt{P}_\emptyset\langle \langle \rangle, \texttt{S}_\emptyset\langle \langle \rangle, \texttt{S}_1\langle \langle \rangle, \langle \rangle \rangle \rangle \rangle, \langle \rangle, \texttt{S}_\emptyset\langle \langle \rangle, \texttt{S}_1\langle \langle \rangle, \langle \rangle \rangle \rangle \rangle \rangle \in \texttt{2CC}$. This single factoring duplicated all ingredients except $\langle \rangle$ and $\langle \rangle$, which where in the factored choice. We now have to factor $\langle \rangle$ with the next choice $\texttt{S}_\emptyset$ and then again at $\texttt{S}_1$ duplicating the ingredients each time. At the end, all ingredients will be at the bottom of the expression forming variants such as $\langle \rangle, \langle \rangle$. Since leaves in algebraic decision trees store only exactly one atom, each leaf node has to reference an entire variant sub-tree as its atom:

$e_6 = \texttt{P}_\emptyset\langle \texttt{S}_\emptyset\langle \texttt{leaf}\ \langle \rangle, \texttt{S}_1\langle \texttt{leaf}\ \langle \rangle, \texttt{leaf}\ \langle \rangle \rangle, \texttt{S}_\emptyset\langle \texttt{leaf}\ \langle \rangle, \texttt{S}_1\langle \texttt{leaf}\ \langle \rangle, \texttt{leaf}\ \langle \rangle \rangle \rangle \rangle \in \texttt{ADT}$.

We now have constructed a circle of choice-based languages $\texttt{CCC} \twoheadrightarrow n\text{-}\texttt{CC} \twoheadrightarrow \texttt{2CC} \twoheadrightarrow \texttt{ADT}$ and back again, where the backwards direction is given by definition. We thus conclude that all these languages are equally expressive.

Corollary 5.5. $\texttt{CCC} \equiv n\text{-}\texttt{CC} \equiv \texttt{2CC} \equiv \texttt{ADT}, \forall n \in \mathbb{N}, n \geq 2$.

5.2 Soundness and Completeness by Clone-and-Own

To prove completeness and soundness of the languages within our circle, we have to do so for at least one language directly. As a vehicle to simplify these direct proofs, we use non-empty lists of variants $(v_1, \ldots, v_n) \in \underline{e}^n, n \in \mathbb{N}$. In software development, treating variants individually like this is common practice (e.g., using branching or forking) and known as clone-and-own (CaO) [81, 88, 89, 91, 146]. CaO constitutes a most basic variability language, unable to encode any decision process or similarities, where configurations are indices $i \in \mathbb{N}$ and the semantics is a lookup $[\![(v_1, \ldots, v_n)]\!](i) = v_{min(i,n)}$. Removing these degrees of freedom make CaO an easy target for direct proofs of completeness and soundness. By integrating CaO into our circle, we conclude completeness and soundness for the other languages inside the circle for free (cf. Section 4.4).

Theorem 5.6. Complete(CaO).

Proof Sketch. Given a variant generator $V : \mathbb{N}_n \to \mathbb{V}(A), n \in \mathbb{N}$ over an atom set A, we construct $l = (V(1), \ldots, V(n)) \in \texttt{CaO}$. To prove $[\![l]\!] \cong V$ we translate configurations as follows: $\text{conf} : \mathbb{N} \to \mathbb{N}_n$ with $\text{conf}(i) := min(i, n)$ and $\text{fnoc} : \mathbb{N}_n \to \mathbb{N}$ with $\text{fnoc}(i) := i$. Then $\forall i \in \mathbb{N} : [\![l]\!](i) = V(\text{conf}(i)) = V(min(i, n))$ and $\forall i \in \mathbb{N}_n : [\![l]\!](\text{fnoc}(i)) = [\![l]\!](i) = V(i)$ by definition. □

Theorem 5.7. Sound(CaO).

Proof Sketch. By definition, all lists $e \in \texttt{CaO}$ are non-empty and finite, hence denoting a variant generator, hence being sound. A formal proof involves computing the variant generator explicitly by showing that any list is bounded in its length, and that a list enumerates its infinitely many configurations $\mathbb{N}$ w.r.t. configuration equivalence (cf. Section 4.4). □

Theorem 5.8. $\texttt{CaO} \succeq \texttt{ADT}$.

Proof Sketch by Compilation. Let $e \in \texttt{ADT}$. We create $\text{t}(e) \in \texttt{CaO}$ by collecting the leaf variants from left to right: $\text{t}(\text{leaf}\ v) = [v]$ and $\text{t}(D\langle l, r\rangle) = \text{t}(l) +\!\!+ \text{t}(r)$. Since t may include dead variants (e.g., in $D\langle D\langle ☺, ☹\rangle, ☺\rangle$, ☹ is dead because reaching it would require to set D to *true* and *false* simultaneously), we construct an intra-language compiler for dead-branch elimination (step 1). We

then must bimap configurations $\mathbb{N}$ for CaO to $\mathbb{F} \to \mathbb{B}$ for ADT, which we do in two steps. First, we bimap $\mathbb{F} \to \mathbb{B}$ to paths from the root to leaves and define alternative ADT semantics $[\![.]\!]_p$ based on paths (step 2). Second, we bimap paths to $\mathbb{N}$ by interpreting a path as a binary search (step 3). We prove 1-3 by induction and conclude $[\![e]\!]_{\mathsf{ADT}} \cong [\![\mathsf{t}(e)]\!]_{\mathsf{CaO}}$ transitively. □

Example 5.9. Translating our previous example e_6 we get the list $e_7 = \mathsf{t}(e_6) = ($  ,  ,  ,  ,  , ), which contains all variants from left to right. Since e_6 does not contain any dead branches, the dead branch elimination has no effect here. Indexing e_7 corresponds to walking a path on e_6. For example, $[\![e_7]\!](2) =$  $= [\![e_6]\!]_p((\mathsf{P}_\emptyset, \mathit{true}) :: (\mathsf{S}_\emptyset, \mathit{false}) :: (\mathsf{S}_1, \mathit{true}) :: [\,])$.

Theorem 5.10. CCC $\succeq$ CaO.

Proof Sketch by Compilation. Let $(v_1, \ldots, v_n) \in \mathsf{CaO}, n \in \mathbb{N}$. We create $D\langle v_1, \ldots, v_n\rangle \in \mathsf{CCC}$. Proving correctness is straightforward because indices in both expressions remain the same. □

Translating CaO to CCC in fact embodies an n-way tree differencing algorithm (cf. Section 4.4). The output expression may display differences across variants as choices and similarities via shared atoms. Proving the existence of a correct compiler requires only any correct differencer, not a good one. Hence, we simply declare all trees to be different via a single big choice.

We now exploit the circle of expressiveness for choice-based languages to obtain completeness and soundness for free for all respective languages via Theorem 4.22 and Theorem 4.23.

Corollary 5.11. CCC, 2CC, ADT, *and* n−CC *are complete and sound,* $\forall n \in \mathbb{N}, n \geq 2$.

5.3 Option Calculus

Having covered choice-based languages, our interest shifts towards *option*-based variability and option calculus (OC). The key question here is whether options and choices are equally expressive.

First, we show that choices can indeed model options. Typically, choice-based formalisms encode options via neutral atoms [14, 94, 126, 144] (cf. Section 3.6). However, neutral atoms might not exist and depend on context (i.e., the monoid of the enclosing expression if there is a monoid at all). For example, in variational propositional logic [144], the neutral value is *true* for conjunctions $\wedge$ but *false* for disjunctions $\vee$. As observed by Gruler [64, p. 45], neutral values in a choice can be eliminated by factoring: $[\![P \oplus_i (P \parallel R)]\!] \cong [\![P \parallel (\mathsf{ntrl} \oplus_i R)]\!]$ means that a choice between P and $P \parallel R$ is equivalent to composing P with an option over R, encoded as a choice $\mathsf{ntrl} \oplus_i R$ with a neutral value ntrl. As an example, the OC expression  $\prec$Salad(),  $\succ$, which denotes a sandwich with Cheese and optionally Salad, can be expressed in 2CC as  $\prec$Salad$\langle$  , $\varepsilon\rangle$,  $\succ$, where ε denotes an empty but pointless sandwich ingredient, which distributes to Salad$\langle$  $\prec$  ,  $\succ$,  $\prec$  $\succ\rangle$. Hence, an option can be turned into choice without the need of neutral atoms.

Theorem 5.12. 2CC $\succeq$ OC.

Proof Sketch by Compilation. Let $e \in \mathsf{OC}$. By definition $e = a\prec x_1, \ldots, x_n\succ$. We translate the children $x_i \in \mathsf{OC}$ sequentially from left to right. When we encounter an artifact $x_i = b\prec \ldots \succ$, we recursively translate x_i and then proceed with x_{i+1}. When we encounter an option $x_i = O\langle y\rangle$, we fork the translation in two branches. The first branch includes the expression y by replacing x_i with y and then continues from there, eventually yielding a translated expression $e_y \in \mathsf{2CC}$. The second branch discards the expression y and continues translation with x_{i+1}, eventually yielding a translated expression $e_{\neg y} \in \mathsf{2CC}$. We then introduce a choice $O\langle e_y, e_{\neg y}\rangle \in \mathsf{2CC}$ as result. Configurations remain constant and correctness can be proven by induction. The formal proof requires an intermediate language to keep track of translated sub-expressions $x_j, j < i$. □

Corollary 5.13. Sound(OC).

To investigate whether options may also express choices, we first have a look at completeness. We already observed in Section 3.6 that we could not encode our sandwich example in OC because we could not specify  and  to be alternative.

Theorem 5.14. ¬Complete(OC).

Proof by Contradiction. Assume Complete(OC). Let $V : \mathbb{N}_2 \to \mathbb{V}(\{$  ,  $\})$ be a variant generator with $V(1) =$  , $V(2) =$  . By completeness, there exists $e \in \mathsf{OC}$ with $[\![e]\!] \cong V$. Thus, there exist configurations c_1, c_2 with  $= [\![e]\!](c_1)$ and  $= [\![e]\!](c_2)$. By definition $e = a\prec \ldots \succ, a \in \{$  ,  $\}$. If $a =$  , c_2 cannot exist because configuring e will always yield  $\prec \ldots \succ$. Analogously, if $a =$  , c_1 cannot exist. Hence, our assumption $[\![e]\!] \cong V$ is violated. □

Given that 2CC is complete but OC is not, there cannot exist a compiler from 2CC to OC, and we can cross-out the respective edge in Figure 3. Hence, 2CC, and by transitivity also all other choice-based languages on the left in Figure 3, are more expressive than OC.

Theorem 5.15. $\mathsf{2CC} \succ \mathsf{OC}$.

Proof. By Theorem 5.12, we know $\mathsf{2CC} \succeq \mathsf{OC}$. It is left to show that the opposite $\mathsf{OC} \succeq \mathsf{2CC}$ does not hold (i.e., $\mathsf{OC} \not\succeq \mathsf{2CC}$). Assume $\mathsf{OC} \succeq \mathsf{2CC}$. We conclude Complete(OC) from Complete(2CC) (Corollary 5.11) via Theorem 4.22, which violates ¬Complete(OC) (Theorem 5.14). □

5.4 Feature Structure Trees

Having covered annotative languages, we now turn to feature structure trees (FST) as compositional variability language (cf. Section 3.7). We observe that FST is incomplete for at least two reasons.

Theorem 5.16. ¬Complete(FST).

Proof Sketch. Every expression $a \blacktriangleleft f_1, \ldots, f_n \blacktriangleright \in \mathsf{FST}$ has an atom a at the top. Hence, FST is incomplete for the same reason OC is incomplete for (cf. Theorem 5.14). Furthermore, neighboring nodes $b_1\prec \ldots \succ, \ldots, b_n\prec \ldots \succ$ in a feature must not have duplicate atoms ($b_i \neq b_j, \forall i \neq j$) [11]. Hence, FST cannot represent a variant violating this restriction such as $a\prec b, b\succ$. □

Despite both FST and OC being incomplete, they have different expressiveness.

Theorem 5.17. $\mathsf{OC} \not\succeq \mathsf{FST}$.

Proof Sketch. Let A be an atom set. There cannot exist a compiler $\mathsf{FST} \twoheadrightarrow \mathsf{OC}$ by counterexample $e = a \blacktriangleleft X : b\prec c\succ, Y : b\prec d\succ \blacktriangleright \in \mathsf{FST}; a, b, c, d \in A; c \neq d; X, Y \in \mathbb{F}$. We find $[\![e]\!]$ is a variant generator of the following variants: (1) a for $X = Y = \mathit{false}$, (2) $a\prec b\prec c\succ\succ$ for $X = \mathit{true}, Y = \mathit{false}$, (3) $a\prec b\prec d\succ\succ$ for $X = \mathit{false}, Y = \mathit{true}$, (4) $a\prec b\prec c, d\succ\succ$ for $X = Y = \mathit{true}$. Assuming there exists a compiler $\mathsf{FST} \twoheadrightarrow \mathsf{OC}$, we can translate e into an option calculus expression $e' = a\prec t_1, \ldots, t_n\succ$ describing exactly the above set of variants. Variant (2) implies that there exists i and c_1 such that $[\![t_i]\!](c_1) = b\prec c\succ$. Variant (3) implies that there exists j and c_2 such that $[\![t_j]\!](c_2) = b\prec d\succ$. If $i \neq j$, then the artifact $[\![e']\!](\lambda x.\ \mathit{true})$ contains at least two top level children t_i and t_j. However, e does not encode any such variant, hence $i = j$. Let $c_\wedge(O) = c_1(O) \wedge c_2(O), \forall O \in \mathbb{F}$. Then $[\![t_i]\!](c_\wedge) = b$ because b must be at the top of t_i and the sub-terms c and d are excluded in one of the variants each, and hence are excluded by $c_\wedge$. Hence $[\![e']\!](c_\wedge) = a\prec b\succ$, but e and hence e' do not encode this variant, so $i = j$ is impossible. We obtain a contradiction $i = j$ and $i \neq j$. □

Theorem 5.18. $\mathsf{FST} \not\succeq \mathsf{OC}$.

Proof Sketch. Let A be an atom set. There cannot exist a compiler $\texttt{OC} \rightarrowtriangle \texttt{FST}$ by counterexample $e = a\prec b, b\succ \in \texttt{OC}$, $a, b \in A$. FST features must not have neighboring artifacts with the same atom and hence cannot encode e (cf. proof of Theorem 5.16). □

As we did for OC, we can conclude from incompleteness that FST is less expressive than any complete language and hence no respective translation can exist.

Corollary 5.19. $\texttt{FST} \not\succeq \texttt{CaO}$.

Theorem 5.20. $\texttt{CaO} \succeq \texttt{FST}$.

Proof Sketch by Compilation. Let $e = a \blacktriangleleft f_1, \ldots, f_n \blacktriangleright \in \texttt{FST}$ with $f_i = F_i : t_i$. A list of all variants is now given by $([\![e]\!](c_1), \ldots, [\![e]\!](c_{2^n}))$ where $c_j(F_i) \coloneqq \mathit{true}$ iff the i-th bit of j, interpreted as a binary number, is set, and $c_j(F_i) \coloneqq \mathit{false}$ otherwise (i.e., we enumerate all possible configurations). In case of duplicate features $F_i = F_{i'}$, c_j only considers the smallest index and ignores the bits corresponding to higher indexes. CaO configurations $\mathbb{N}$, and FST configurations $\mathbb{F} \to \mathbb{B}$ are mapped to each other in terms of the index $j \in \mathbb{N}$ of $c_j : \mathbb{F} \to \mathbb{B}$. Correctness follows from the fact that the semantics of FST never evaluate a configuration for non-existing features, so the CaO expression cannot miss any variants. □

We can now conclude Sound(FST) by Theorem 4.22 and Theorem 5.6.

Corollary 5.21. Sound(FST).

5.5 Discussion and Conclusion

We have now charted a region on the map of variability languages. We found a class of languages that are complete, sound, and equally expressive with respect to our semantic domain, namely CCC, 2CC, n-CC, ADT, and CaO. In particular, we are the first to formally prove the choice calculi to be equally expressive, which has been implicitly assumed in many works [14, 31, 37, 78, 79, 94, 126, 137, 143, 144]. This also means that languages that mix choices with other variational constructs [27, 80, 139], such as options, are also complete and equally expressive (as long as these languages have denotational semantics according to Definition 4.13). The same applies to languages and mechanisms specialized to certain domains or use cases, such as for the Linux kernel.

Surprisingly, we also discovered that there are languages that are incomplete regarding variant generators, namely OC and FST. Crucially, these languages hence cannot be used to statically specify all variational systems imaginable. For instance, there are sandwich menus and sets of Linux kernels which cannot be statically specified in these languages. We find that incompleteness stems from three syntactic restrictions.

First, OC and FST must have a constant atom at the root of all variants. In fact, this restriction was a deliberate design decision we made for option calculus. Dropping this restriction would make option calculus and feature structure trees unsound, because it might produce questionable empty variants, as discussed in Section 3.6, suggesting the existence of a trade-off between completeness and soundness. We argue that soundness is more important because it ensures that a language is meaningful and can be compared to other languages. In practice, a fixed root is probably reasonable because variational systems typically have a common base implementation, root directory, or similar anchor such as  , and such an atom could be artificially introduced. After all, variants are supposed to have at least some degree of shared atoms for variational treatment to make sense.

Second, OC and FST cannot encode mutual exclusion which causes incompleteness. Adding mutual exclusion to OC and FST requires to encode constraints externally or to enrich the language. External constraints are common practice in software product line engineering and covered by problem space variability in terms of a satisfiability challenge for configurations (cf. Section 2). To

enable a language for mutual exclusion, it must be able to react to selection as well as deselection of a feature. We could add choices or `else` branches [27], or instead replace mere names $\mathbb{F}$ as annotations by a more sophisticated *annotation language*, for example propositional formulas [27, 69], but being able to negate names is enough. For example, an option calculus dialect without forced atom root and with negations in its annotation language could express a choice $D\langle l, r\rangle$ as $D(l)(\neg D)(r)$. This dialect however must be constrained to have either an atom or such a pair of options at the top to be sound. Given that alternatives are common in practice, we thus recommend choices or to document configuration constraints in the first place, which brings additional benefits [7, 23, 129].

Third, `FST` are restricted to never have duplication at the same granularity [11] causing incompleteness. Technically, this restriction is necessary to avoid self-composition: For example, composing a feature $e = 1{\prec}2{\succ} :: 1{\prec}3{\succ} :: [\,]$ causes both sub-trees $1{\prec}2{\succ}$, $1{\prec}3{\succ}$ to be imposed on each other: $[\,] \oplus e = ([\,] \odot 1{\prec}2{\succ}) \odot 1{\prec}3{\succ} = [1{\prec}2, 3{\succ}] \neq e$. This means that a sandwich cannot have  directly below twice, or that a Java class cannot have two methods with the same name and signature. In practice, this restriction is reasonable for a range of target languages [5, 9, 11, 19], including popular programming languages, but not for example XML [11].

The existence of such restrictions suggests the existence of different classes of expressiveness — evidenced by option calculus and feature structure trees not being as expressive as each other despite being incomplete. In fact, we found at least three classes of expressiveness: Annotative choice-based languages are complete and hence maximally expressive, while pure options and tree-based composition are not. However, options avoid duplication and ambiguity as discussed in Section 5.3; an important concern for some applications, including variational analyses [14, 144].

Finding different classes of expressiveness because of syntactic restrictions, begs the question as to what the impact of other restrictions may be and what classes of expressiveness might emerge. We might find more classes by relaxing or imposing other restrictions, perhaps by extending the framework with other properties, such as other notions of completeness. A useful property for example, could be *core*-completeness to check whether a language is complete apart from having a top-level atom or other fixed atoms, also known as *core* features [7, 39]. We hence strive to further explore syntactic constraints and their impact on expressiveness in the future.

The map of languages we discover, also comes with practical results for researchers and language designers. First, researchers may now apply research results or tools formulated in a language L to all languages M with less or equal expressive power ($L \succeq M$) by first translating M to L. For instance, with the choice calculus having served as a *lingua franca* in many research efforts (cf. Section 3.3), we may now apply research results such as variational type-checking [79], variational type inference [37], or variational SAT solving [144] to all other languages in our map as well. While not being an end-to-end tooling, our Agda framework, which formalizes our map in terms of proven-to-be correct compilers, can be used, for example, to translate datasets for such purposes. Second, language designers and researchers may leverage our framework for basic sanity checks such as soundness or completeness; and they can relate new languages to existing ones with only one or two translations with respective proofs without having to compare to each language individually (similarly as we did by building compiler circles). Third, in this regard, our map of the language space also provides guidance to future researchers on which language to pick. For example, formalizing and verifying a variational analysis might be easier with algebraic decision trees but a tool for empirical studies might favor choice calculus because it more accurately reflects real implementations (e.g., C preprocessor). With our framework, this is proven possible and correct.

6 Related Work

While we discussed related work on variability languages in Section 3, we cover work on expressiveness in other domains here.

Comparing the Expressive Power of Other Formal Systems. Similar to our framework for variability, expressive power has also been studied for other formal systems, including but not limited to programming languages [41, 44, 56, 106, 109, 122], string constraints [43], type systems [111], or deep neural networks [114]. Some define expressiveness similarly as we do, as a language's ability to denote elements in its semantic domain [41, 43, 56].

In principle, expressiveness can be studied on a syntactic or semantic level. Syntactic comparisons of the expressive power of programming languages [44, 56] emerged from the desire to compare different combinations of language constructs at a finer level of granularity, and in a way perhaps more relevant to human programmers, than semantic comparisons can capture. For example, many programming languages are Turing-complete and so equivalently expressive from a semantic perspective. Yet there are differences with respect to how concisely and with how much redundancy two languages can express semantically equivalent programs. Notably, comparisons of expressive power of *concurrent* programming languages [44] consider choices as language constructs to denote different possible program states during concurrent execution [3, 34], emphasizing the fundamental nature of choices as a notation for multiple states. Syntactic comparisons of variation constructs, especially their ability to support sharing of equal sub-expressions, is in fact highly relevant. As a step towards that goal, we focused on previously unexplored semantic expressiveness first to answer *what* languages can say before tackling the question *how concise* they can do so.

Comparing the Expressive Power of Languages for Problem Space Variability. In this paper, we compare languages for solution space variability (i.e., how to implement variability and derive variants). There have been similar efforts for languages for problem space variability (i.e., constraining configurations, cf. Section 2). Informal comparisons cover specific formats supported by widely-used tools in research and practice [20, 51, 55]. Formal comparisons of problem space languages cover algebraic specifications for translating constraints [68, 85] and powerful compilation formats to optimize solving time [29, 30, 42, 90, 104, 123, 128]. Similarly as we did for solution space, Schobbens et al. [119] defined expressiveness and completeness for feature diagrams. While the languages covered by the works above are related to our work in terms of addressing variational concerns, they are inherently different in that they express constraints, not variants.

Comparing the Expressive Power of Languages for Solution Space Variability. Existing efforts in this direction remain informal or domain-specific, for example focusing on behavioral models [21, 133] or model checking [49]. Walkingshaw [136] briefly highlights that choices can encode options with neutral atoms, and Gruler [64] model options via choices but do not discuss expressiveness. Later, Walkingshaw and Ostermann [137] briefly and informally discuss the expressive power of options versus choices. We proved that choices can express options even without assuming neutral atoms, and develop a generic framework without assumptions on the semantics of variants to study variability as a phenomenon by itself.

7 Conclusion

In this work, we began the journey to chart the space of languages for static variability by introducing a formal framework to study and compare variability languages. We identified a common semantic domain, formalized the semantics of a range of existing languages, established the basic properties of soundness, completeness, and expressiveness, and studied these properties for existing formal languages. We formalize our framework, proofs, and study in Agda as a reusable library including a web of proven-to-be correct compilers and differencing algorithms. By extending the library with a compiler from or to an integrated language, language designers can compare the expressiveness of their language to others, deriving proofs of soundness or completeness for free.

To our surprise, we find multiple levels of expressiveness as well as complete and incomplete languages in the identified semantic domain, arising from different syntactical restrictions. In

particular, choice-based formal languages are complete, sound, and equally expressive, and hence the same is true for choice-based languages used in practice, including the variability mechanisms for the Linux kernel for example. We hence verified previously implicit assumptions and bridged the gap between parallel research efforts.

Data-Availability Statement

Our Agda library, called Vatras, in which we implement the languages from our overview (cf. Section 3), our framework and its proofs (cf. Section 4), and the compilers and proofs for our exploration of variability languages (cf. Section 5), is open source and publicly available online on Github and Zenodo [1].

Acknowledgments

We thank Sebastian Krieter, Clemens Dubslaff, Chico Sundermann, and Sascha Rechenberger for helpful discussions, advice, and pointers to related work. We also thank Chico Sundermann and Larissa Förster for proof-reading and Chico Sundermann, Elias Kuiter, and Sebastian Krieter for testing our replication package. We also thank our reviewers as well as the reviewers for our artifact submission for their detailed and constructive feedback. We thank the God-Emperor for protecting us from the abominations of the warp. This work has been supported by the German Research Foundation (project *VariantSync*, TH 2387/1-1, TH 2387/1-2, KE 2267/1-1) and the Swiss National Science Foundation (project *VariantSync*, grant 222903) as well as CNPq (grants 315532/2021-1 and 423125/2021-4), CAPES (88881.512952/2020-01), Alexander von Humboldt Foundation, and INES[8] (CNPq grant 465614/2014-0, CAPES grant 88887.136410/2017-00, and FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03/14).

References

[1] 2024. Vatras – Our Supplementary Agda Library. Zenodo DOI: 10.5281/zenodo.13502454, Github: https://github.com/VariantSync/Vatras.

[2] Karthik. V. Aadithya, Tomasz P. Michalak, and Nicholas R. Jennings. 2011. Representation of Coalitional Games With Algebraic Decision Diagrams. In *Proc. Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*. International Foundation for Autonomous Agents and Multiagent Systems, 1121–1122.

[3] Alejandro Aguirre and Lars Birkedal. 2023. Step-Indexed Logical Relations for Countable Nondeterminism and Probabilistic Choice. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, POPL, Article 2 (2023), 28 pages.

[4] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf H. Reussner. 2022. A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications. *Empirical Software Engineering (EMSE)* 27, 5 (2022), 101.

[5] Felipe I. Anfurrutia, Oscar Díaz, and Salvador Trujillo. 2007. On Refining XML Artifacts. In *Proc. Int'l Conf. on Web Engineering (ICWE)*, Luciano Baresi, Piero Fraternali, and Geert-Jan Houben (Eds.), Vol. 4607. Springer, 473–478.

[6] Sven Apel. 2007. *The Role of Features and Aspects in Software Development.* Ph. D. Dissertation. University of Magdeburg.

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

[8] Sven Apel, Christian Kästner, and Don Batory. 2008. Program Refactoring Using Functional Aspects. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 161–170.

[9] Sven Apel, Christian Kästner, and Christian Lengauer. 2013. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. on Software Engineering (TSE)* 39, 1 (2013), 63–79.

[10] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. 2005. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. Springer, 125–140.

[11] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. 2010. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming (SCP)* 75, 11 (2010), 1022–1047.

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

[12] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. 2006. An Overview of CaesarJ. *Trans. Aspect-Oriented Software Development* 1 (2006), 135–173.
[13] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2012. A Compositional Framework to Derive Product Line Behavioural Descriptions. In *Proc. Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Springer, 146–161.
[14] Parisa Ataei, Fariba Khan, and Eric Walkingshaw. 2021. A Variational Database Management System. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 29–42.
[15] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proc. Symposium on Principles of Programming Languages (POPL)*. ACM, 165–178.
[16] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1993. Algebraic Decision Diagrams and Their Applications. In *Proc. Int'l Conf. on Computer-Aided Design (ICCAD)*. IEEE, 188–191.
[17] Michael Barr and Charles Wells. 1985. *Toposes, Triples and Theories*. Springer.
[18] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20.
[19] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering (TSE)* 30, 6 (2004), 355–371.
[20] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual Variability Modeling Languages: An Overview and Considerations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 151–157.
[21] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. 2016. Basic Behavioral Models for Software Product Lines: Expressiveness and Testing Pre-Orders. *Science of Computer Programming (SCP)* 123, C (2016), 42–60.
[22] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. 2005. Classbox/J: Controlling the Scope of Change in Java. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 177–189.
[23] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Modeling in the Real: A Perspective From the Operating Systems Domain. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 73–82.
[24] Paul Maximilian Bittner, Alexander Schultheiß, Sandra Greiner, Benjamin Moosherr, Sebastian Krieter, Christof Tinnes, Timo Kehrer, and Thomas Thüm. 2023. Views on Edits to Variational Software. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 141–152.
[25] Paul Maximilian Bittner, Alexander Schultheiß, Benjamin Moosherr, Timo Kehrer, and Thomas Thüm. 2024. Variability-Aware Differencing with DiffDetective. In *Companion Proc. Int'l Conference on the Foundations of Software Engineering (FSE Companion)*. ACM, 632–636.
[26] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 1007–1020.
[27] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. 2022. Classifying Edits to Variability in Source Code. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 196–208.
[28] Yuri Breitbart, H Hunt III, and Daniel Rosenkrantz. 1995. On the Size of Binary Decision Diagrams Representing Boolean Functions. *Theoretical Computer Science* 145, 1-2 (1995), 45–69.
[29] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers* C-35, 8 (1986), 677–691.
[30] Randal E. Bryant. 2018. Binary Decision Diagrams. In *Handbook of Model Checking*. Springer, 191–217.
[31] John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2022. Migrating Gradual Types. *Journal of Functional Programming (JFP)* 32 (2022). https://doi.org/10.1017/S0956796822000089
[32] Mikaela Cashman, Myra B. Cohen, Priya Ranjan, and Robert W. Cottingham. 2018. Navigating the Maze: The Impact of Configurability in Bioinformatics Software. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 757–767.
[33] Thiago Castro, Leopoldo Teixeira, Vander Alves, Sven Apel, Maxime Cordy, and Rohit Gheyi. 2021. A Formal Framework of Software Product Line Analyses. *Trans. on Software Engineering and Methodology (TOSEM)* 30, 3, Article 34 (2021), 37 pages.
[34] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, POPL, Article 61 (2023), 31 pages.
[35] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 362–374.

[36] Sheng Chen. 2015. *Variational Typing and Its Applications.* Ph. D. Dissertation. Oregon State University.

[37] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems (TOPLAS)* 36, 1, Article 1 (2014), 1:1–1:54 pages.

[38] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 335–344.

[39] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley.

[40] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 23–34.

[41] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys (CSUR)* 33, 3 (2001), 374–425.

[42] Adnan Darwiche and Pierre Marquis. 2002. A Knowledge Compilation Map. *J. Artificial Intelligence Research (JAIR)* 17, 1 (2002), 229–264.

[43] Joel D. Day, Vijay Ganesh, Nathan Grewal, and Florin Manea. 2023. On the Expressive Power of String Constraints. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, POPL, Article 10 (2023), 31 pages.

[44] Frank S. de Boer and Catuscia Palamidessi. 1991. Embedding as a Tool for Language Comparison: On the CSP Hierarchy. In *Proc. Int'l Conf. on Concurrency Theory (CONCUR)*, Vol. 527. Springer, 127–141.

[45] Nicolaas Govert De Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, With Application to the Church-Rosser Theorem. *Indagationes Mathematicae* 75, 5 (1972), 381–392.

[46] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. 2017. Efficient Family-Based Model Checking via Variability Abstractions. *Int'l J. Software Tools for Technology Transfer (STTT)* 19, 5 (2017).

[47] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2018. Variability Abstractions for Lifted Analyses. *Science of Computer Programming (SCP)* 159 (2018), 1–27.

[48] Clemens Dubslaff. 2019. Compositional Feature-Oriented Systems. In *Proc. Int'l Conf. on Software Engineering and Formal Methods (SEFM)*, Peter Csaba Ölveczky and Gwen Salaün (Eds.). Springer, 162–180.

[49] Clemens Dubslaff. 2022. *Quantitative Analysis of Configurable and Reconfigurable Systems.* Ph. D. Dissertation. Dresden University of Technology.

[50] Jeffrey Dudek, Vu Phan, and Moshe Vardi. 2020. ADDMC: Weighted Model Counting With Algebraic Decision Diagrams. In *Proc. Conf. on Artificial Intelligence (AAAI)*, Vol. 34. AAAI Press, 1468–1476.

[51] Holger Eichelberger and Klaus Schmid. 2015. Mapping the Design Space of Textual Variability Modeling Languages: A Refined Analysis. *Int'l J. Software Tools for Technology Transfer (STTT)* 17, 5 (2015), 559–584.

[52] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 45–54.

[53] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *Trans. on Software Engineering and Methodology (TOSEM)* 21, 1, Article 6 (2011), 27 pages.

[54] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. 313–324.

[55] Kevin Feichtinger, Chico Sundermann, Thomas Thüm, and Rick Rabiser. 2022. It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 67–78.

[56] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming (SCP)* 17, 1-3 (1991), 35–75.

[57] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 316–326.

[58] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2014. A Taxonomy of Software Product Line Reengineering. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 4:1–4:8.

[59] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 665–668.

[60] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. on Software Engineering (TSE)* 33, 11 (2007), 725–743.

[61] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *Proc. Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 91–100.

[62] Steven J Friedman and Kenneth J Supowit. 1990. Finding the Optimal Variable Ordering for Binary Decision Diagrams. *IEEE Trans. on Computers* 39, 5 (1990), 710–713.

[63] Paul Gazzillo and Myra B. Cohen. 2022. Bringing Together Configuration Research: Towards a Common Ground. In *Proc. Int'l Symposium on New Ideas, New Paradigms, and Reflections on Programming (Onward!)*. ACM, 259–269.
[64] Alexander Gruler. 2010. *A Formal Approach to Software Product Families.* Ph. D. Dissertation. TU München.
[65] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. 2008. Modeling and Model Checking Software Product Lines. In *Proc. IFIP Int'l Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Springer, 113–131.
[66] Alireza Haghighatkhah, Ahmad Banijamali, Olli-Pekka Pakanen, Markku Oivo, and Pasi Kuvaja. 2017. Automotive Software Engineering: A Systematic Mapping Study. *J. Systems and Software (JSS)* 128 (2017), 25–55.
[67] Ruidong Han, Chao Yang, Siqi Ma, JiangFeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. 2022. Control Parameters Considered Harmful: Detecting Range Specification Bugs in Drone Configuration Modules via Learning-Guided Search. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 462–473.
[68] Peter Höfner, Ridha Khédri, and Bernhard Möller. 2011. An Algebra of Product Families. *Software and Systems Modeling (SoSyM)* 10, 2 (2011), 161–182.
[69] Spencer Hubbard and Eric Walkingshaw. 2016. Formula Choice Calculus. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 49–57.
[70] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability With Embedded Annotations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 61–70.
[71] Ralph E. Johnson and Brian Foote. 1988. Designing Reusable Classes. *J. of Object-Oriented Programming (JOOP)* 1, 2 (1988), 22–35.
[72] Seiede Reyhane Kamali, Shirin Kasaei, and Roberto E. Lopez-Herrejon. 2019. Answering the Call of the Wild? Thoughts on the Elusive Quest for Ecological Validity in Variability Modeling. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 143–150.
[73] 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 CMU/SEI-90-TR-21. Software Engineering Institute.
[74] Christian Kästner. 2010. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* Ph. D. Dissertation. University of Magdeburg.
[75] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 311–320.
[76] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-Based Product Lines. *Trans. on Software Engineering and Methodology (TOSEM)* 21, 3 (2012), 14:1–14:39.
[77] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. 2009. Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, Manuel Oriol and Bertrand Meyer (Eds.). Springer, 175–194.
[78] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 805–824.
[79] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-Aware Module System. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 773–792.
[80] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-Aware Testing. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 1–8.
[81] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 21–25.
[82] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *Proc. Europ. Conf. on Object-Oriented Programming (ECOOP)*. Springer, 327–354.
[83] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *Proc. Europ. Conf. on Object-Oriented Programming (ECOOP)*. Springer, 220–242.
[84] Gregor Kiczales and Mira Mezini. 2005. Aspect-Oriented Programming and Modular Reasoning. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 49–58.
[85] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 291–302.
[86] Emily Kowalczyk, Myra B. Cohen, and Atif M. Memon. 2018. Configurations in Android Testing: They Matter. In *Proc. Int'l Workshop on Advances in Mobile App Analysis (A-Mobile)*. ACM, 1–6.
[87] Christian Kröher, Lea Gerling, and Klaus Schmid. 2023. Comparing the Intensity of Variability Changes in Software Product Line Evolution. *J. Systems and Software (JSS)* 203 (2023), 111737.

[88] Jacob Krüger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 21, 10 pages.

[89] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 432–444.

[90] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 110:1–110:13.

[91] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-and-Own: Moving to a Software Product Line for Temperature Monitoring. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 179–189.

[92] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming (SCP)* 78, 8 (2013), 1010–1034.

[93] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 105–114.

[94] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.

[95] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 49–62.

[96] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2016. A Variability Aware Configuration Management and Revision Control Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 803–806.

[97] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *Software and Systems Modeling (SoSyM)* 16, 4 (2017), 1179–1199.

[98] Jia Liu, Don Batory, and Christian Lengauer. 2006. Feature Oriented Refactoring of Legacy Applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 112–121.

[99] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1658–1670.

[100] Tabassum Mahmud, Om Rameshwar Gatla, Duo Zhang, Carson Love, Ryan Bumann, and Mai Zheng. 2023. ConfD: Analyzing Configuration Dependencies of File Systems for Fun and Profit. In *USENIX Conf. on File and Storage Technologies (FAST)*. USENIX Association, 199–214.

[101] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. 2001. Jiazzi: New-Age Components for Old-Fasioned Java. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 211–222.

[102] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 483–494.

[103] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 231–240.

[104] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient Compilation Techniques for Large Scale Feature Models. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 13–22.

[105] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2015. Systematic Derivation of Correct Variability-Aware Program Analyses. *Science of Computer Programming (SCP)* 105 (2015), 145–170.

[106] John C. Mitchell. 1993. On Abstraction and the Expressive Power of Programming Languages. *Science of Computer Programming (SCP)* 21, 2 (1993), 141–163.

[107] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 907–918.

[108] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 893–905.

[109] Joachim Parrow. 2008. Expressiveness of Process Algebras. *Electronic Notes in Theoretical Computer Science (ENTCS)* 209 (2008), 173–186.

[110] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. *Empirical Software Engineering (EMSE)* 21, 4 (2016).

[111] Marco Patrignani, Eric Mark Martin, and Dominique Devriese. 2021. On the Semantic Expressiveness of Recursive Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 5, POPL, Article 21 (2021), 29 pages.
[112] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants With VariantSync. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 329–332.
[113] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.
[114] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. 2017. On the Expressive Power of Deep Neural Networks. In *Proc. Int'l Conf. on Machine Learning (ICML)*, Vol. 70. PMLR, 2847–2854.
[115] Kamil Rosiak and Ina Schaefer. 2023. The e4CompareFramework: Annotation-based Software Product-Line Extraction. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 34–38.
[116] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 101–110.
[117] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 77–91.
[118] Thomas Schmorleiz and Ralf Lämmel. 2016. Similarity Management of 'Cloned and Owned' Variants. In *Proc. ACM Symposium on Applied Computing (SAC)*. ACM, 1466–1471.
[119] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007), 456–479.
[120] Felix Schwägerl and Bernhard Westfechtel. 2019. Integrated Revision and Variation Control for Evolving Model-Driven Software Product Lines. *Software and Systems Modeling (SoSyM)* 18, 6 (2019), 3373–3420.
[121] Ramy Shahin and Marsha Chechik. 2020. Automatic and Efficient Variability-Aware Lifting of Functional Programs. *Proceedings of the ACM on Programming Languages (PACMPL)* 4, OOPSLA, Article 157 (2020), 27 pages.
[122] Ehud Shapiro. 1989. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys (CSUR)* 21, 3 (1989), 413–510.
[123] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 620–636.
[124] Yannis Smaragdakis and Don Batory. 2002. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *Trans. on Software Engineering and Methodology (TOSEM)* 11, 2 (2002), 215–255.
[125] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. 2002. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Proc. Int'l Conf. on Technology of Object-Oriented Languages and Systems (TOOLS)*. Australian Computer Society, 53–60.
[126] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 323–333.
[127] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering (EMSE)* 28, 29 (2023), 38.
[128] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. 2023. On the Benefits of Knowledge Compilation for Feature-Model Analyses. *Annals of Mathematics and Artificial Intelligence (AMAI)* (2023).
[129] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 12, 10 pages.
[130] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2024. Reusing d-DNNFs for Efficient Feature-Model Counting. *Trans. on Software Engineering and Methodology (TOSEM)* (2024). To appear.
[131] GNU Operating System. 2024. Autoconf. Website. Available online at https://www.gnu.org/software/autoconf/; visited on April 2nd, 2024..
[132] GNU Operating System. 2024. GNU M4. Website. Available online at https://www.gnu.org/software/m4/; visited on April 2nd, 2024..
[133] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2019. On the Expressiveness of Modal Transition Systems With Variability Constraints. *Science of Computer Programming (SCP)* 169 (2019), 1–17.
[134] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6:1–6:45.
[135] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book.

[136] Eric Walkingshaw. 2013. *The Choice Calculus: A Formal Language of Variation.* Ph. D. Dissertation. Oregon State University.
[137] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE).* ACM, 29–38.
[138] Ingo Wegener. 1987. *The Complexity of Boolean Functions.* Springer.
[139] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. 2016. Custom-Tailored Variability Mining for Block-Based Languages. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 271–282.
[140] Chu-Pan Wong. 2021. *Beyond Configurable Systems: Applying Variational Execution to Tackle Large Search Spaces.* Ph. D. Dissertation. Carnegie Mellon University.
[141] Chu-Pan Wong, Jens Meinicke, and Christian Kästner. 2018. Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2018).* ACM, 749–753.
[142] Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution With Transparent Bytecode Transformation. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA, Article 117 (2018), 30 pages.
[143] Jeffrey M. Young. 2021. *Variational Satisfiability Solving.* Ph. D. Dissertation. Oregon State University.
[144] Jeffrey M. Young, Paul Maximilian Bittner, Eric Walkingshaw, and Thomas Thüm. 2022. Variational Satisfiability Solving: Efficiently Solving Lots of Related SAT Problems. *Empirical Software Engineering (EMSE)* 28 (2022), 53.
[145] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. 2021. An Evolutionary Study of Configuration Design and Implementation in Cloud Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE).* IEEE, 188–200.
[146] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *Proc. Int'l Conf. on Software Engineering (ICSE).* ACM, 105–116.

Received 2024-04-05; accepted 2024-08-18