Variational Satisfiability Solving

Jeffrey M. Young
youngjef@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

Eric Walkingshaw
walkiner@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

Thomas Thüm
thomas.thuem@uni-ulm.de
University of Ulm
Ulm, Germany

ABSTRACT

Incremental satisfiability (SAT) solving is an extension of classic SAT solving that allows users to efficiently solve a set of related SAT problems by identifying and exploiting shared terms. However, using incremental solvers effectively is hard since performance is sensitive to a problem's structure and the order sub-terms are fed to the solver, and the burden to track results is placed on the end user. For analyses that generate sets of related SAT problems, such as those in software product lines, incremental SAT solvers are either not used at all, used but not explicitly stated so in the literature, or used but suffer from the aforementioned usability problems. This paper translates the ordering problem to an encoding problem and automates the use of incremental SAT solving. We introduce *variational* SAT solving, which differs from incremental SAT solving by accepting all related problems as a single variational input and returning all results as a single variational output. Our central idea is to make explicit the operations of incremental SAT solving, thereby encoding differences between related SAT problems as local points of variation. Our approach automates the interaction with the incremental solver and enables methods to automatically optimize sharing of the input. To evaluate our methods we construct a prototype variational SAT solver and perform an empirical analysis on two real-world datasets that applied incremental solvers to software evolution scenarios. We show, assuming a variational input, that the prototype solver scales better for these problems than naive incremental solving while also removing the need to track individual results.

CCS CONCEPTS

• **Software and its engineering** → *Software product lines*; • **Hardware** → **Theorem proving and SAT solving**;

KEYWORDS

satisfiability solving, variation, choice calculus, software product lines

ACM Reference Format:

Jeffrey M. Young, Eric Walkingshaw, and Thomas Thüm. 2020. Variational Satisfiability Solving. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3382025.3414965

SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada

ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414965

1 INTRODUCTION

Satisfiability solving is a ubiquitous technology in software product lines for a diverse set of analyses ranging from anomaly detection [2, 38, 46], dead code analysis [62], sampling [47, 65], and automated analysis of feature models [10, 32, 64]. The general pattern is to represent parts of the system or feature model as a propositional formula [9, 25, 48], and reduce the analysis to a satisfiability (SAT) problem. However, modern software is constantly evolving and thus the translation step to a single SAT problem quickly becomes a translation to a set of SAT problems.

Sets of SAT problems frequently arise, for example, when analyzing changes to feature models over time. Consider a feature model for some product version i, represented as a conjunction of clauses that describe the relationships among features: $FM_i = c_0 \wedge c_1 \ldots\ c_n$. One might perform a single analysis (e.g., dead feature analysis) over several versions or commits yielding a set of SAT problems (clauses that are altered from version FM_i are underlined):

$$\begin{aligned} SAT_{FM_i} &= (c_0 \wedge\ c_1 \wedge\ c_2 \wedge\ c_3 \ldots c_n) \wedge dead\_feat \\ SAT_{FM_{i+1}} &= (c_0 \wedge\ \underline{c_1} \wedge\ c_2 \wedge\ \underline{c_3} \ldots c_n) \wedge dead\_feat \\ &\vdots \\ SAT_{FM_{i+n}} &= (\underline{c_0} \wedge\ c_1 \wedge\ c_2 \wedge\ c_3 \ldots \underline{c_n}) \wedge dead\_feat \end{aligned}$$

Or consider a case where several properties must be guaranteed for every commit via a continuous integration tool:

$$\begin{aligned} SAT_{FM_i\_\text{void}} &= (c_0 \wedge\ c_1 \wedge\ c_2 \wedge\ c_3 \ldots \wedge c_n) \\ SAT_{FM_i\_\text{core}} &= ((c_0 \wedge\ c_1 \wedge\ c_2 \wedge\ c_3 \ldots \wedge c_n) \wedge \neg core\_feat) \\ &\vdots \\ SAT_{FM_{i+n}\_\text{core}} &= ((\underline{c_0} \wedge\ c_1 \wedge\ c_2 \wedge\ c_3 \ldots \wedge \underline{c_n}) \wedge \neg other\_core\_feat) \end{aligned}$$

In such cases, state-of-the-art methods do not make use of commonalities among the set of formulas, perform redundant computation, and lose learned information from previous SAT calls.

A concrete example of the above scenario involves the Linux Foundation's response to the meltdown and spectre security vulnerabilities [37, 45]. The response resulted in three kinds of Linux kernel versions and three corresponding feature models: a model that does not support exploit prevention features, a version that supports several exploit prevention features but not a single, global toggle, and a version that aggregates all prevention features to a single feature. The different kernel versions were used throughout the software industry, and many companies, such as cloud service providers, employed products that simultaneously used each version. Hence any SAT-based analysis on such products would lead to a set of SAT problems, with one problem per supported kernel.

Analyzing such products thus leads to analyses over sets of SAT problems, where performing an analysis over each feature model becomes inefficient: We must either perform the analysis on each feature model individually, thus not making any use of apriori known commonalities, or try to reuse results by running the analysis on the feature model with no prevention features, and apply the results to feature models that have some prevention features. However, such a plan is spurious; changes between kernel versions could have introduced significant cross-tree constraints that would not be captured by reuse, and reusing results would require domain knowledge and a high degree of manual effort.

An alternative is to use an incremental SAT solver, which allows the user to hand-write a program to consider shared terms only once, then direct the solver to solutions, one for each feature model, in the search space. This is more efficient because it reuses knowledge of shared terms, however, using an incremental SAT solver in this way requires substantial manual effort and domain knowledge, it produces a specific solution to a specific analysis, and it requires extra infrastructure to manage results.

Our solution is to formalize a method of satisfiability solving that makes use of known commonalities among propositional formulas and automates the interaction with an incremental SAT solver, thus providing efficiency benefits while reducing the usability problems to an encoding problem. Our central idea is to translate the implicit operations of incremental SAT solving into a static representation that makes obvious the terms in the input formula that can change (those indexed by one or several formulas) and terms that are not subject to change (those shared among formulas). We call this method *variational* satisfiability solving because it understands and efficiently handles queries that differentiate between terms that are constant with respect to a set of propositional formulas (i.e., *plain* terms), and terms that are subject to change (i.e., *variational* terms).

Our approach has many benefits: (1) End-users are only required to provide a single *variational formula*, which represents a set of related propositional formulas, rather than a formula *and* a hand-written program to direct the solver. (2) It is general; while variational satisfiability solving is applied to feature model analyses in this work, it can be used for any analysis that can be encoded as a variational formula. (3) With a variational formula, new kinds of syntactic manipulations, such as factoring out shared terms, become possible and can be automated. (4) A *variational model* may be produced that encapsulates a set of satisfying assignments for all variants of the variational formula, alleviating the need to track the incremental solver's results when satisfying assigments are needed.

We describe the process of variational SAT solving and the construction of variational models in Section 4, and construct a prototype solver based on these ideas. We evaluate performance with a variational void analysis, and demonstrate a variational dead and core feature analysis. We perform these analyses on two variational formulas, which represent four and ten versions of two real world software artifacts' feature models. For this work, we focus only on variational satisfiability solving and assume a variational formula as input, leaving other considerations such as the optimal encoding of such formulas to future work. Our contributions are as follows:

- We give the syntax and semantics of an extension to propositional logic that reasons about variation. (Section 3)
- We design and implement variational models. (Section 4)
- We present an algorithm that solves formulas in the extended logic using off-the-shelf incremental solvers as black boxes. The prototype solver is publicly available.[1] (Section 4)
- We report a performance improvement over standard methods when solving many variants, and demonstrate variational void, core, and dead feature analyses. (Section 5.2)

2 BACKGROUND

Variational SAT solving depends on incremental SAT solving. In this section, we describe the underlying data structures and operations that variational satisfiability solving exploits, using the Linux Kernel as the running example. Our description, and the interface between variational SAT solving and incremental SAT conforms to the SMTLIB2 [8] standard.

After the discovery of the meltdown and spectre security vulnerabilities, there were multiple versions of the Linux kernel that dealt with these vulnerabilities (or not) in different ways. Suppose, for example, we have kernel versions L_0, L_1, and L_2 with corresponding feature models FM_0, FM_1, and FM_2. FM_0 contains no spectre/meltdown-related features; FM_1 contains a set of new features named `spectre_v2`, `nospec_store_bypass_disable`, `l1tf`, and `pti`; and FM_2 contains a single feature `mitigations` that combines all of the exploit prevention features from FM_1.[2]

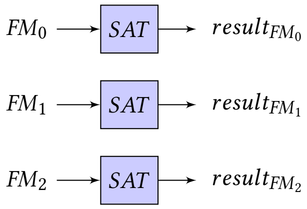
We introduce some notation to track particular features and propositional formulas across multiple feature models. For features we use $f_{i.j}$ to refer to the ith feature in the jth feature model. For formulas, we use $c_{i.j}$ to refer to the formula that encodes the ith feature's relationships to other features in the jth feature model. When the feature model version is omitted, e.g., c_i, we assume that c_i is unchanged and present in all feature models. Thus, the feature models can be represented by the following formulas:

$$\begin{aligned}
FM_0 &= c_{0.0} \wedge c_{1.0} \wedge \ldots \wedge c_n \\
FM_1 &= (\underline{spectre_v2} \vee \underline{l1tf}) \leftrightarrow (\underline{c_{0.0}} \wedge (\underline{nospec_store_bypass\text{-}disable} \\
&\quad \rightarrow \underline{f_j}) \wedge c_{1.0} \wedge (\underline{pti} \rightarrow \underline{c_{i.0}}) \wedge \ldots \wedge c_n) \\
FM_2 &= \underline{mitigations} \leftrightarrow (c_{0.0} \wedge c_{1.0} \wedge \ldots \wedge c_n)
\end{aligned}$$

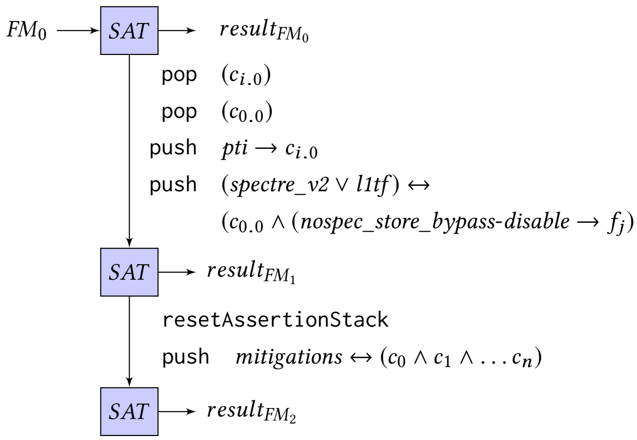
FM_0 is a conjunction of formulas that describe the relationship of features in L_0. In FM_1 we can see exactly how several clauses have been changed. New features have been introduced, e.g., *pti*, $c_{0.0}$ is constrained with a new conjunction, and there are three new formulas: $(pti \rightarrow c_{i.0})$, $(spectre\_v2 \vee l1tf)$, $(nospec\_store\_bypass\text{-}disable \rightarrow f_j$, two of which affect a relationship or feature from FM_0. In FM_2, the features and constraints introduced in FM_1 are replaced by a single new *mitigations* feature that is added to an unchanged copy of FM_0.

Suppose one wants to find a satisfying assignment (i.e., a model) for each formula. If done with a classic SAT solver, then the procedure illustrated in Figure 1a results; where SAT solving is a batch process and no information is reused. Alternatively, a procedure using an incremental SAT solver is illustrated in Figure 1b; in this scenario, all of the formulas are solved by single solver instance where terms are programmatically added and removed from the solver throughout the process. The ability to add and remove terms

[1] https://github.com/lambda-land/VSat-Papers/tree/master/SPLC2020
[2] The feature names are from the actual Linux kernel, see [42].



(a) Brute force procedure, no reuse between solver calls.



(b) Incremental procedure, reuse defined by pop and push calls.

Figure 1

from the solvers is enabled by a data structure within the incremental SAT solver called an *assertion stack*. The assertion stack is a stack of declarations, definitions, or formulas that determine the *context* of the solver. A solver context is the union of all global variable definitions and everything on the assertion stack. A program may add an assertion to the stack via the `push` operation and remove from the top via a `pop` operation [51].

In an efficient process one would initially add as many shared terms as possible, FM_0 in this example. Then request a model, and manipulate the assertion stack to reach the next problem of interest, FM_1 in this case. Notice that to reach the next problem, FM_1, from FM_0, several operations are required: $c_{0.0}$ and $c_{i.0}$ must be removed, $c_{0.0}$ must be updated, and the new sub-formulas must be introduced. To reach FM_2 from FM_1 all assertions would need to be popped to add *mitigation*, then re-pushed.

3 VPL: VARIATION + PROPOSITIONAL LOGIC

In this section, we present the logic of variational satisfiability problems. The logic is a conservative extension of classic two-valued logic (C_2) with a *choice* construct from the choice calculus [29, 68], a formal language for describing variation. We call the new logic VPL, short for variational propositional logic, and refer to VPL expressions as *variational formulas*. This section defines the syntax and semantics of VPL and uses it to encode the example from Section 2.

Syntax. The syntax of variational propositional logic is given in Figure 2a. It extends the propositional formula notation of C_2 with a single new connective called a *choice* from the choice calculus. A choice $D\langle f_1, f_2\rangle$ represents either f_1 or f_2 depending on the Boolean value of its *dimension* D. We call f_1 and f_2 the *alternatives*

$$
\begin{array}{rcll}
t & ::= & r \mid \mathsf{T} \mid \mathsf{F} & \textit{Variables and Boolean literals} \\[4pt]
f & ::= & t & \textit{Terminal} \\
& \mid & \neg f & \textit{Negate} \\
& \mid & f \vee f & \textit{Or} \\
& \mid & f \wedge f & \textit{And} \\
& \mid & D\langle f, f\rangle & \textit{Choice}
\end{array}
$$

(a) Syntax of VPL.

$$
\begin{aligned}
[\![\cdot]\!] &: f \to C \to f \qquad\qquad \text{where } C = D \to \mathbb{B}_\perp \\
[\![t]\!]_C &= t \\
[\![\neg f]\!]_C &= \neg [\![f]\!]_C \\
[\![f_1 \wedge f_2]\!]_C &= [\![f_1]\!]_C \wedge [\![f_2]\!]_C \\
[\![f_1 \vee f_2]\!]_C &= [\![f_1]\!]_C \vee [\![f_2]\!]_C \\
[\![D\langle f_1, f_2\rangle]\!]_C &= \begin{cases} [\![f_1]\!]_C & C(D) = \mathsf{true} \\ [\![f_2]\!]_C & C(D) = \mathsf{false} \\ D\langle [\![f_1]\!]_C, [\![f_2]\!]_C\rangle & C(D) = \perp \end{cases}
\end{aligned}
$$

(b) Configuration semantics of VPL.

$$
\begin{array}{rcll}
D\langle f, f\rangle & \equiv & f & \textsc{Idemp} \\
D\langle D\langle f_1, f_2\rangle, f_3\rangle & \equiv & D\langle f_1, f_3\rangle & \textsc{Dom-L} \\
D\langle f_1, D\langle f_2, f_3\rangle\rangle & \equiv & D\langle f_1, f_3\rangle & \textsc{Dom-R} \\
D_1\langle D_2\langle f_1, f_2\rangle, D_2\langle f_3, f_4\rangle\rangle & \equiv & D_2\langle D_1\langle f_1, f_3\rangle, D_1\langle f_2, f_4\rangle\rangle & \textsc{Swap} \\
D\langle \neg f_1, \neg f_2\rangle & \equiv & \neg D\langle f_1, f_2\rangle & \textsc{Neg} \\
D\langle f_1 \vee f_3,\ f_2 \vee f_4\rangle & \equiv & D\langle f_1, f_2\rangle \vee D\langle f_3, f_4\rangle & \textsc{Or} \\
D\langle f_1 \wedge f_3,\ f_2 \wedge f_4\rangle & \equiv & D\langle f_1, f_2\rangle \wedge D\langle f_3, f_4\rangle & \textsc{And} \\
D\langle f_1 \wedge f_2, f_1\rangle & \equiv & f_1 \wedge D\langle f_2, \mathsf{T}\rangle & \textsc{And-L} \\
D\langle f_1 \vee f_2, f_1\rangle & \equiv & f_1 \vee D\langle f_2, \mathsf{F}\rangle & \textsc{Or-L} \\
D\langle f_1, f_1 \wedge f_2\rangle & \equiv & f_1 \wedge D\langle \mathsf{T}, f_2\rangle & \textsc{And-R} \\
D\langle f_1, f_1 \vee f_2\rangle & \equiv & f_1 \vee D\langle \mathsf{F}, f_2\rangle & \textsc{Or-R}
\end{array}
$$

(c) VPL equivalence laws

Figure 2: Formal definition of VPL.

of the choice. Although dimensions are Boolean variables, the set of dimensions is disjoint from the set of variables from C_2, which may be referenced in the leaves of a formula. We use lowercase letters to range over variables and uppercase letters for dimensions.

The syntax of VPL does not include derived logical connectives, such as $\rightarrow$ and $\leftrightarrow$. However, such forms can be defined from other primitives and are assumed throughout the paper.

Semantics. Conceptually, a variational formula represents several propositional logic formulas at once, which can be obtained by resolving all of the choices. For software product-line researchers, it is useful to think of VPL as analogous to `#ifdef`-annotated C_2, where choices correspond to a disciplined [43] application of `#ifdef` annotations. From a logical perspective, following the many-valued logic of Kleene [56], the intuition behind VPL is that a choice is a placeholder for two equally possible alternatives that is deterministically resolved by reference to an external environment. In this sense, VPL deviates from other many-valued logics, such as modal logic [33], because a choice *waits* until there is enough information to choose an alternative (i.e., until the formula is *configured*).

The *configuration semantics* of VPL is given in Figure 2b and describes how choices are eliminated from a formula. The semantics is parameterized by a *configuration* C, which is a partial function from dimensions to Boolean values. The first four cases of the semantics simply propagate configuration down the formula, terminating at the leaves. The case for choices is the interesting one: if the dimension of the choice is defined in the configuration, then the choice is replaced by its left or right alternative corresponding to the associated value of the dimension in the configuration. If the dimension is undefined in the configuration, then the choice is left intact and configuration propagates into the choice's alternatives.

If a configuration C eliminates all choices in a formula f, we call C *total* with respect to f. If C does *not* eliminate all choices in f (i.e., a dimension used in f is undefined in C), we call C *partial* with respect to f. We call a choice-free formula *plain*, and call the set of all plain formulas that can be obtained from f (by configuring it with every possible total configuration) the *variants* of f.

To illustrate the semantics of VPL, consider the formula $p \wedge A\langle q, r\rangle$, which has two variants: $p \wedge q$ when $C(A) = \text{true}$ and $p \wedge r$ when $C(A) = \text{false}$. From the semantics, it follows that choices in the same dimension are *synchronized* while choices in different dimensions are *independent*. For example, $A\langle p, q\rangle \wedge B\langle r, s\rangle$ has four variants, while $A\langle p, q\rangle \wedge A\langle r, s\rangle$ has only two ($p \wedge r$ and $q \wedge s$). It also follows from the semantics that nested choices in the same dimension contain redundant alternatives; that is, inner choices are *dominated* by outer choices in the same dimension. For example, $A\langle p, A\langle r, s\rangle\rangle$ is equivalent to $A\langle p, s\rangle$ since the alternative r cannot be reached by any configuration. As the previous example illustrates, the representation of a VPL formula is not unique; that is, the same set of variants may be encoded by different formulas. Figure 2c defines a set of equivalence laws for VPL formulas. These laws follow directly from the configuration semantics in Figure 2b and can be used to derive semantics-preserving transformations of VPL formulas. For example, we can simplify the formula $A\langle p \vee q, p \vee r\rangle$ by first applying the Or law to obtain $A\langle p, p\rangle \vee A\langle q, r\rangle$, then applying the Idemp law to the first argument to obtain $p \vee A\langle q, r\rangle$ in which the redundant p has been factored out of the choice.

Running example. To demonstrate the application of VPL, we encode the evolving Linux kernel feature model from the background as a variational formula. Recall that variation in this domain arises from changes in the logical structure of the feature model between kernel versions. Our goal is to construct a single variational formula that encodes the set of all feature models as variants. Ideally, this variational formula should also maximize sharing among the feature models in order to avoid redundant analysis later.

Every set of plain formulas can be encoded as a variational formula systematically by first constructing a nested choice containing all of the individual variables as alternatives, then factoring out shared subexpressions by applying the laws in Figure 2c. For sets of feature models this would correspond to a nested choice containing all of the individual feature models as alternatives, then factoring out commonalities in the variational formula. Unfortunately, the process of globally minimizing a variational formula in this way is hard[3] since often we must apply an arbitrary number of laws right-to-left in order to set up a particular sequence of left-to-right applications that factor out commonalities.

Due to the difficulty of minimization, we instead demonstrate how one can build such a formula *incrementally*. Our variational formula will use the dimensions $L_1, \ldots, L_n$ to refer to changes introduced in the feature model in the corresponding version of the Linux kernel. We begin by combining FM_0 and FM_2 since the differences between the two are smaller than between other pairs of feature models in our example. Feature models may be combined in any order as long as the variants in the resulting formula correspond to their plain counterparts. The only change between FM_0 and FM_2 is the addition of *mitigations* and is captured by a choice in dimension L_2. The change is nested in the left alternative so that it will be included for any configuration where L_2 is true. This yields the following variational formula.

$$f_{FM_{02}} = L_2\langle \mathit{mitigations}, \top\rangle \leftrightarrow c_{0.0} \wedge c_1 \wedge \ldots \wedge c_n$$

We exploit the fact that $\wedge$ forms a monoid with $\top$ to recover a formula equivalent to FM_0 for configurations where L_2 is false.

Next we combine $f_{FM_{02}}$ with FM_1 to obtain a variational formula that captures the feature models of versions L_0, L_1, and L_2. As before, every change in FM_1 is wrapped in a choice in dimension L_1. The choice in L_2 is nested in the right alternative of a choice in L_1 because that change is not present in L_1:

$$\begin{aligned} f_{FM_{012}} = {} & L_1\langle(\mathit{spectre\_v2} \vee \mathit{l1tf}), L_2\langle \mathit{mitigations}, \top\rangle\rangle \\ & \leftrightarrow L_1\langle(c_{0.0} \wedge (\mathit{nospec\_store\_bypass\text{-}disable} \rightarrow f_j), c_{0.0}\rangle \\ & \wedge L_1\langle c_{1.0}, \top\rangle \wedge c_1 \wedge L_1\langle(\mathit{pti} \rightarrow c_{i.1}), \top\rangle \wedge \ldots \wedge c_n \end{aligned}$$

Now that we have constructed the variational formula we need to ensure that it encodes all variants of interest and nothing else. In this example, this is relatively easy to confirm by enumerating all total configurations involving L_1 and L_2. However, we'll return to the general case in the discussion of variational models in Section 4.

4 VARIATIONAL SATISFIABILITY SOLVING

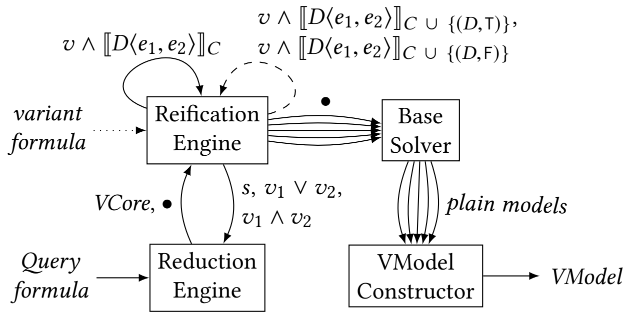
In this section, we provide an informal description of variational satisfiability solving and variational models. A formal semantics is available in an online appendix.[4] Throughout the section, we use SMTLIB2 snippets to describe variational solving concepts in terms of an incremental solver. While we target SMTLIB2, conforming to the standard is not a requirement. Any solver that exposes an incremental API as defined by minisat [51] can be used to implement variational satisfiability solving.

We use a recursive approach to solve a VPL formula, decoupling the handling of plain terms from the handling of variational terms. The idea is to define a process to evaluate plain terms and skip choices, then define another process that only configures choices thus introducing new plain terms to the formula that can be recursively processed. The base case is a variant, at which point a model can be queried and the assertion stack can be popped to backtrack to solve another variant.

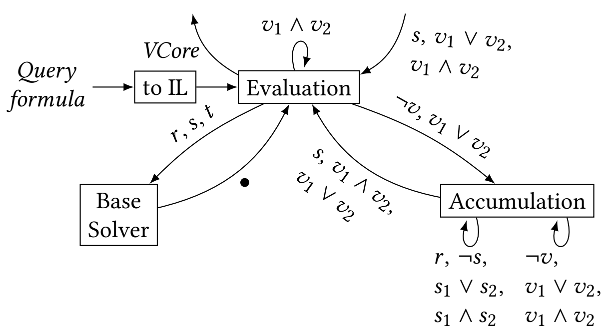
We present an overview of a variational solver as a state diagram in Figure 3a that operates on the input's abstract syntax tree. Labels

[3] We hypothesize that it is equivalent to BDD minimization, which is NP-complete, but the equivalence has not been proved; see [69].

[4] https://github.com/lambda-land/VSat-Papers/tree/master/SPLC2020



(a) System overview of a variational solver.



(b) Overview of the reduction engine.

Figure 3

on incoming edges denote inputs to a state and labels on outgoing edges denote return values; we show only inputs for recursive edges; labels separated by a comma share the edge. We omit labels that can be derived from the logical properties of connectives, such as commutativity of $\vee$ and $\wedge$. Similarly, we omit base case edge labels for choices and describe these cases in the text. The solver has four subsystems: The *reduction engine* processes plain terms and generates a formula ready for reification called a *variational core*. The *reification engine* configures choices in a variational core. The *base solver* is the incremental solver used to produce plain models. Finally, the *variational model constructor* synthesizes a single variational model from the set of plain models returned by the base solver.

The solver takes a VPL formula, called a *query formula*, and an optional input, called a *variation context* (*vc*). A *vc* is a propositional formula of dimensions that restricts the solver to a subset of variants. The variational solver translates the query formula to a formula in an intermediate language (IL) that the reduction and reification engines operate over; its syntax is given below.

$$v \ ::= \ \bullet \mid t \mid s \mid \neg v \mid v \wedge v \mid v \vee v \mid D\langle e, e\rangle$$

The IL includes two kinds of terminals not present in the input query formulas: plain sub-terms that can be reduced symbolically will be replaced by a *symbolic reference* s, and sub-terms that have been sent to the base solver will be represented by the unit value $\bullet$. Note that choices contain unprocessed expressions (e) as alternatives.

Derivation of a Variational Core. A variational core is an IL formula that captures the variational structure of a query formula. Plain terms will either be placed on the assertion stack or will be symbolically reduced, leaving only logical connectives, symbolic references, and choices. Consider the query formula $f = ((a \wedge b) \wedge A\langle e_1, e_2\rangle) \wedge ((p \wedge \neg q) \vee B\langle e_3, e_4\rangle)$. Translated to an IL formula, f has four references (a, b, p, q) and two choices. The reduction engine shown in Figure 3b will produce a variational core that will assert $(a \wedge b)$ in the base solver, thus pushing it onto the assertion stack and create a symbolic reference for $(p \wedge \neg q)$. This is done in two states: *evaluation*, which communicates to the base solver to process plain terms, and *accumulation* which is called by evaluation to create symbolic references.

Generating the core begins with evaluation. Evaluation will match on the root node: $\wedge$, of f and recur following the $v_1 \wedge v_2$ edge, where $v_1 = (a \wedge b) \wedge A\langle e_1, e_2\rangle$ and $v_2 = (p \wedge \neg q) \vee B\langle e_3, e_4\rangle$. The recursion processes the left child first. Thus, evaluation will again match on $\wedge$ of v_1 creating another recursive call with $v_1' = (a \wedge b)$ and $v_2' = A\langle e_1, e_2\rangle$. Finally, the base case is reached with a last recursive call where $v_1'' = a$, and $v_2'' = b$. At the base case both a and b are references, thus evaluation will send a to the base solver, following the r, s, t edge, which returns $\bullet$ for the left child. The right child follows the same process yielding $\bullet \wedge \bullet$; since the assertion stack implicitly conjuncts all assertions, $\bullet \wedge \bullet$ will be further reduced to $\bullet$ and returned as the result of v_1', indicating that both children have been pushed to the base solver. This leaves $v_1' = \bullet$ and $v_2' = A\langle e_1, e_2\rangle$. v_2' is a base case for choices and cannot be reduced in evaluation, and so $\bullet \wedge A\langle e_1, e_2\rangle$, will be reduced to just $A\langle e_1, e_2\rangle$ as the result for v_1.

In evaluation, conjunctions can be split because of the behavior of the assertion stack and the and-elimination property of $\wedge$. Disjunctions and negations cannot be split in this way because both cannot be performed if a child node has been lost to the solver, e.g., $\neg \bullet$. Thus, in accumulation, we construct symbolic terms to represent entire sub-trees, ensuring information is not lost, but still allowing for the sub-tree to be evaluated if it is sound to do so.

The right child, $v_2 = (p \wedge \neg q) \vee B\langle e_3, e_4\rangle$ requires accumulation. Evaluation will match on the root $\vee$, and send $(p \wedge \neg q) \vee B\langle e_3, e_4\rangle$ to accumulation via the $v_1 \vee v_2$ edge. Accumulation has two self-loops, one to create symbolic references (with labels $r, s, \ldots$), and one to recur to values. Accumulation matches the root $\vee$ and recurs on the self-loop with edge $v_1 \vee v_2$, $v_1 = (p \wedge \neg q)$, and $v_2 = B\langle e_3, e_4\rangle$. Processing the left child first, accumulation will recur again with $v_1' = p$ and $v_2' = \neg q$. $v_1' = p$ is a base case for references, thus a unique symbolic reference s_p is generated for p, following the self-loop with label r and returned as the result for v_1'. v_2' will follow the self-loop with label $\neg v$ to recur through $\neg$ to q, where a symbolic term s_q will be generated and returned. This yields $\neg s_q$, which follows the $\neg s$ edge to be processed into a new symbolic term, yielding the result for v_2' as $s_{\neg q}$. With both results $v_1 = s_p \wedge s_{\neg q}$, accumulation will match on $\wedge$ *and* both s_p and $s_{\neg q}$ to accumulate the entire sub-tree to a single symbolic term, $s_{s_p \wedge s_{\neg q}}$, which will be returned as the result for v_1. v_2 is a base case, hence accumulation will return $s_{s_p \wedge s_{\neg q}} \vee B\langle e_3, e_4\rangle$ to evaluation. Evaluation will conclude with $A\langle e_1, e_2\rangle$ as the result for the left child of $\wedge$ and $s_{s_p \wedge s_{\neg q}} \vee B\langle e_3, e_4\rangle$ for the right child, yielding $A\langle e_1, e_2\rangle \wedge s_{s_p \wedge s_{\neg q}} \vee B\langle e_3, e_4\rangle$ as the variational core of f.

A variational core is derived to save redundant work. If solved naively, plain sub-formulas of f, such as $a \wedge b$ and $p \wedge \neg q$, would be processed once for each variant even though they are unchanged. Evaluation moves sub-formulas into the solver state to be reused among different variants. Accumulation caches sub-formulas that cannot be immediately evaluated to be evaluated later.

Symbolic references are variables in the reduction engine's memory that represent a set of statements in the base solver. For example, s_{pq} represents three declarations in the base solver:

```
(declare-const p Bool)                   ;; s_pq represents
(declare-const q Bool)                   ;; several declarations
(declare-fun s_ab () Bool (or p (not c)))
```

Similarly a variational core is a sequence of statements in the base solver with holes $\Diamond$. For example, the representation of $VCore_f$:

```
(assert (and a b))                       ;; a ∧ b on the assertion stack
(declare-const ◇)                        ;; choice A
⋮                                        ;; many declares may occur
(assert ◇)                               ;; many assertions may occur
⋮                                        ;; s_pq
(declare-fun s_pq () Bool (and p q))
(declare-const ◇)                        ;; choice B
⋮
(assert (or s_ab ◇))                     ;; assert waiting on ⟦B⟨e3, e4⟩⟧_C
```

Each hole is filled by configuring a choice and may require multiple statements to process the alternative.

Solving the Variational Core. The reduction engine performs the work at each recursive step. Whereas the reification engine defines transitions between the recursive steps by manipulating the configuration. In VPL, a configuration was formalized as a function, for variational solvers we use a set of tuples $\{(D \times \mathbb{B})\}$. Figure 3a shows two self-loops for the reification engine corresponding to the reification of choices. The edges from the reification engine to the reduction engine are transitions taken after a choice is removed, where new plain terms have been introduced and thus a new core is derived. If the user supplied a variation context, then it is used to construct an initial configuration. Finally, a model is called from the base solver when the reduction engine returns $\bullet$, indicating that a variant has been found.

We display a subset of edges of the reification engine using the $\wedge$ connective. In general, these edges will be duplicated for each binary logical connective, e.g., $\vee$. The left edge, is taken when a choice is observed in the variational core: $v \wedge [\![D\langle e_1, e_2\rangle]\!]_C$ and $D \in C$. This edge reduces choices with dimension D to an alternative, which are then translated to IL. The right edge is dashed to indicate assertion stack manipulation, and is taken when $D \notin C$. For this edge, the configuration is mutated for both alternatives: $C \cup \{(D, \mathsf{T})\}$, and $C \cup \{(D, \mathsf{F})\}$, and the recursive call is wrapped with a push, and pop command. To the base solver, this branching is a linear sequence of assertion stack manipulations that performs backtracking behavior, for example the representation of f is:

```
⋮              ;; declares and assertions from VCore
(push 1)       ;; a configuration on B has occurred
⋮              ;; new declarations for left alternative
(declare-fun s () Bool (or s_pq ◇[◇ → s_B_T]))  ;; fill
(assert s)
⋮              ;; recursive processing
(pop 1)        ;; return for the right alternative
(push 1)       ;; repeat for right alternative
```

Where the hole $\Diamond$, will be filled with a newly defined variable s_{D_T} that represents the left alternative's formula.

$$
\begin{array}{lll}
f_0 \to \mathsf{T} & f_0 \to \mathsf{T} & f_0 \to \mathsf{T} \\
\vdots & \vdots & \vdots \\
f_i \to \mathsf{F} & f_i \to \mathsf{T} & f_i \to \mathsf{F} \\
\vdots & \vdots & \vdots \\
f_n \to \mathsf{F} & f_n \to \mathsf{F} & f_n \to \mathsf{F} \\
C_{FF} = \{(L_1, \mathsf{F}), (L_2, \mathsf{F})\} & mitigations \to \mathsf{T} & nospec\_\ldots \to \mathsf{F} \\
 & C_{FT} = \{(L_1, \mathsf{F}), (L_2, \mathsf{T})\} & spectre_{v2} \to \mathsf{T} \\
 & & l1tf \to \mathsf{T} \\
 & & pti \to \mathsf{F} \\
 & & C_{TT} = \{(L_1, \mathsf{T}), (L_2, \mathsf{T})\}
\end{array}
$$

Figure 4: Possible plain models for variants of $f_{FM_{02}}$.

$$
\begin{array}{l}
\_Sat \to (L_1 \wedge L_2) \vee (\neg L_1 \wedge \neg L_2) \vee (\neg L_1 \wedge L_2) \\
f_0 \to (L_1 \wedge L_2) \vee (\neg L_1 \wedge \neg L_2) \vee (\neg L_1 \wedge L_2) \\
\vdots \\
f_i \to (\neg L_1 \wedge L_2) \\
\vdots \\
f_n \to \mathsf{F} \\
mitigations \to (\neg L_1 \wedge L_2) \\
nospec\_\ldots \to \mathsf{F} \\
spectre_{v2} \to (L_1 \wedge L_2) \\
l1tf \to (L_1 \wedge L_2) \\
pti \to \mathsf{F}
\end{array}
$$

Figure 5: Variational model of the plain models in Figure 4.

Variational Models. Plain models map variables to Boolean values; variational models map variables to variation contexts that record the variants where the variable was assigned T. We denote the variation context for a variable r as vc_r, and maintain a variable called *_Sat* to track which configurations are satisfiable. As an example, consider the query formula ($f_{FM_{012}}$) from the Linux example in Section 2. If each variant is satisfiable, there are three models, as illustrated in Figure 4; the corresponding variational model is shown in Figure 5. $vc_{\_Sat}$ consists of three disjuncted terms, one for each satisfiable variant. A satisfiable assignment of the query formula can be found by calling SAT on $vc_{\_Sat}$. Assuming the model $C_{FT} = \{(L_1, \mathsf{F}), (L_2, \mathsf{T})\}$ is returned, substitution on vc_{f_i} yields f_i's value in C_{FT}:

$$
\begin{array}{lr}
f_i \to (\neg L_1 \wedge L_2) & vc \text{ for } f_i \\
f_i \to (\neg \mathsf{F} \wedge \mathsf{T}) & \text{Substitute } \mathsf{F} \text{ for } L_1, \mathsf{T} \text{ for } L_2 \\
f_i \to \mathsf{T} & \text{Result}
\end{array}
$$

Furthermore, finding variants where a variable f_j is satisfiable reduces to $SAT(vc_{f_j})$

Variational models are constructed incrementally by merging each new plain model returned by the solver into the variational model. A merge requires the current configuration, the plain model, and current *vc* of a variable. Variables are initialized to F. For each variable i in the model, if i's assignment is T in the plain model, then the configuration is translated to a variation context and disjuncted with vc_i. For example, to merge the C_{FT}'s plain model to the variational model in Figure 5, C_{FT}'s configuration is converted to $\neg L_1 \wedge L_2$. This clause is disjuncted for variables assigned T in the plain model: vc_0, vc_i, and $vc_{mitigations}$, even if they are new (e.g., *mitigations*). Variables assigned F are skipped, thus vc_n remains F. In the next model C_{TT}, f_i is F thus vc_i remains unaltered. Variables such as f_n, whose *vc*'s stay F are called *constant*.

5 QUANTITATIVE EVALUATION

Section 4 provides a technique for variational solving that enables sharing work on subterms that are common across several variants. However, the technique also involves substantial overhead, so it is not obvious that it leads to performance gains in realistic problems. To investigate, we construct a prototype variational solver, VSAT in the Haskell programming language [35] and quantitatively compare it to incremental and non-incremental SAT solving. We reuse real-world data from a previous study by Nieke et al. [53]. Nieke et al.'s study provides two datasets, *automotive02* and *financialServices1*, which encode the evolution histories of two feature models as propositional formulas.[5] We refer to these as the *auto* dataset, and *fin* dataset for the remainder of the paper.

5.1 Experimental Methodology

It is important to distinguish between concepts in the application domain, such as a void or core analysis, and concepts in the solver domain, such as a query or choice. When it is potentially ambiguous, we use {brackets} to refer to concepts in the application domain. We use the phrase *version variant* to refer to a variant that is a {version or snapshot} of a sound feature model in the application domain. Choices in different dimensions can be used to encode several different application-domain concepts simultaneously, but they are all interpreted identically in the solver domain.

For example, and to demonstrate the flexibility of variational solving, we construct a VPL formula that encodes both a dead analysis and core analysis over all features f in a query formula q by introducing a choice with a new dimension DC that *does not* correspond to any version: $q\_DC = q \wedge DC\langle \bigwedge_{f \in q} f, \bigwedge_{f \in q} \neg f \rangle$. If q encodes several variants identified by dimensions $V_0, \ldots, V_n$, then q_DC contains dimensions that correspond to two different concepts in the application domain (V_i for versions and DC for the kind of analysis. Selecting an analysis is then performed by a *vc*:[6] $exactly_1(V_0, \ldots, V_n)$. The *vc* selects exactly one version variant with $exactly_1(V_0, \ldots, V_n)$ but leaves the dimension DC undefined. With DC undefined, VSAT will try both DC set to T and DC set to F. Thus, the *vc* selects exactly two variants per version variant, one for the core analysis and one for the dead analysis. To include a void analysis, in addition to the core and dead analyses, another choice is required: $q\_VDC = q \wedge Void\langle DC\langle \bigwedge_{f \in q} f, \bigwedge_{f \in q} \neg f \rangle, \mathsf{T}\rangle$.

We assess the performance characteristics VSAT by attempting to answer the following research questions using our datasets.

RQ1 How does variational solving scale as variation increases?

RQ2 What is the impact of sharing on performance?

RQ3 What is the cost of solving a plain formula on VSAT?

To investigate **RQ1**, we consider all variants of the VPL formulas constructed for each dataset, rather than just the version variants that are of interest in the application domain. This allows us to better evaluate how VSAT scales to accomodate variability. For **RQ2**, we hypothesize that VSAT will get faster as sharing increases, which would validate our method of deriving a variational core. To investigate this, we restrict the analysis to consecutive version variants (i.e., {consecutive monthly snapshots of a feature model}), and observe performance as sharing is left uncontrolled. Finally, **RQ3** provides insight on the overhead incurred by variational solving, which we investigate by inputting each version variant as a propositional logic formula rather than a single variational formula.

Data Description and Encoding. Nieke et al.'s formulas collapse sets of C_2 formulas to a single formula using implications on an SMT variable that represents a moment in time. A two-pass process was used to translate Nieke et al.'s formulas into VPL—one pass to parse to an internal representation and another to detect and convert Nieke et al.'s temporal ranges to choices, nesting the implied clauses into the true alternative. The two-pass process conserves Nieke et al.'s ordering of plain terms and encoding. The two datasets differ in important ways. The *auto* dataset encodes four monthly snapshots while the *fin* dataset encodes ten. Hence, the *auto*'s query formula represents 16 variants, while the *fin* query formula represents 1,024 variants. For **RQ2** and **RQ3**, we construct several *vc*'s to restrict the analysis to version variants. The *vc*s range from ones that enable only one version variant (for **RQ3**): $fmf_{auto\_V_1} = (V_1 \wedge \neg V_2 \wedge \neg V_3 \wedge \neg V_4)$ to *vc*s that enable only consecutive version variants (for **RQ2**): $fmf_{auto\_V12} = V_1 \veebar V_2$.

For **RQ2** we decouple performance from the number of variants by performing an initial pass over the query formula to replace choices representing non-consecutive {versions} with their false alternatives (which contain the value T). Then we constructed a *vc* to forbid non-version variants. As an example, the *auto* dataset, yields three data points by this process, the change from versions V_1 to V_2, V_2 to V_3, and V_3 to V_4.

Measuring Performance. To answer our research questions, we construct four different solving algorithms using our prototype tool. We use the notation *<formula>*⟶*<solver>* to describe, for each algorithm, whether the query formulas and solver are plain (p) or variational (v), respectively. The algorithms are: the baseline, $p{\longrightarrow}p$, which runs plain formulas on a plain solver; variational case, $v{\longrightarrow}v$, which runs a variational formula on the variational solver; the overhead case, $p{\longrightarrow}v$, which runs plain formulas on the variational; and and the exponential case, $v{\longrightarrow}p$, which runs the variational formula on a plain solver. Inputs for each algorithm are constructed by configuring the query formula, thus ensuring that the same variation context is used across algorithms.

We construct the $p{\longrightarrow}p$ algorithm by configuring the query formula to its version variants *before* benchmarking begins. These formulas are then sent to the base solver one-by-one, without the solver maintaining information between queries. To assess the potential overhead of solving a plain query on a variational solver, the $p{\longrightarrow}v$ case and corresponding to **RQ3**, we perform the same

[5] https://gitlab.com/evolutionexplanation/evolutionexplanation

[6] We use a binomial encoding for the exact constraint, see [12, Section 2].

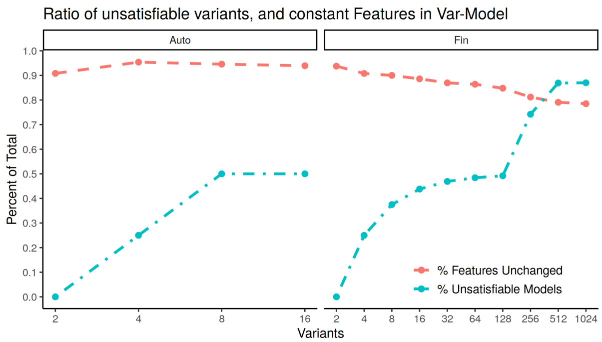


Figure 6: Most models found to be unsatisfiable. Only a small portion of features ever flipped to T.

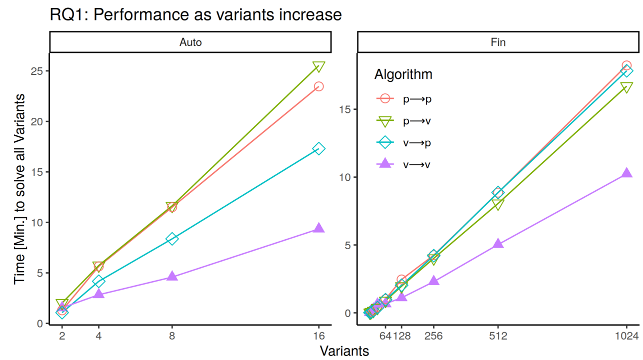


Figure 7: $v \longrightarrow v$ shows a speedup of 2.2-2.5x (*auto*), and 1.84-1.99x (*fin*). Overlapping x-axis labels elided.

pre-processing as the $p \longrightarrow p$ case but send each plain formula to VSAT instead. This provides insight into the cost incurred by the reduction engine. For $v \longrightarrow p$, we configure the query formula to retrieve version variants *during* benchmarking. Each formula is sent to the base solver *with* the solver maintaining information between queries. This gives insight into the overhead incurred by configuring a variational formula and the benefits of caching.

We construct a variational model for all algorithms since it is unclear how to combine plain models otherwise, and since the storage of plain models is an orthogonal concern to performance, we sought to keep convolved variables constant.

Unless specified, all results are a bootstrapped statistical average.[7] For **RQ2**, we normalize the data to the baseline ($v \longrightarrow p$), fit a linear model, and statistically assess differences of samples by performing a one-way Kruskall-Wallis test [54] followed by a pair-wise Wilcox test [70] with Bonferroni p-value correction [27]. For **RQ3**, we retrieve the 10 raw measurements from the bootstrapped average and assess statistical differences identically to **RQ2**. All results, including variational models and statistical analysis scripts, are available online.[8]

5.2 Results and Discussion

Non-performance Results. The datasets yielded dissimilar query formulas: the *auto* query formula consisted of 4,212 choice terms, and 26,808 plain terms. In contrast, *fin* had 3,809 choice terms, and 1,441 plain terms. Thus *fin* had larger changes between {versions}. Figure 6 shows the ratio of unsatisfiable models to total plain models, and the ratio of constant features for each {version} (as represented by variant count). For both datasets the number of satisfiable models decreased as new {versions} were considered, and the majority of features remained constant. Thus, the variational model is likely a compressed version of the set of plain models. Compression metrics were not calculated as this is an orthogonal concern to the performance of variational satisfiability solving.

Variational models permit product analyses without a SAT solver. Figure 6 shows such a purely syntactic analysis: counting disjuncted clauses in the variational model as a representation of satisfiable plain models. We believe post-hoc analyses such as this may be useful to feature modelers as they direct attention to impactful versions of the feature model. For example, the change to V_8 from V_7 (128 to 256 Variants) of *fin* clearly constrained the feature model, decreasing the number of constant features.

RQ1: Performance of Variational Solving as Variation Scales. The VSAT tool outperforms other algorithms as the count of variants to solve increases. Figure 7 shows the time to solve the query formula as variants increase from 2 to 16 for the *auto* dataset, and from 2 to 1,024 for the *fin* dataset. For the *auto* dataset, variational solving is faster at 4 variants, with a speedup of 1.6x while for the *fin* dataset variational solving only becomes performant when solving 64 or more variants, with a speedup of 1.56x. When the query formula represents as many plain SAT problems as possible, we observe a speedup of 2.2x for *auto* and 1.99x for *fin*. However, 87% of results were found to be unsatisfiable for *fin* and 50% for *auto*, thus the performance of variational solving for less constrained formulas remains an open question. Furthermore, we only observed a constant factor speedup; by this data, variational solving still grows linearly in the number of variants being solved.

VSAT outperforms the other algorithms because the variational core caches plain terms, thereby preventing the re-evaluation of these terms for each variant. We observe that derivation of a core only pays off after a particular threshold of the variants to solve is passed. Estimating this threshold value without solving is likely to be important for end-users and so is a topic for future work.

RQ2: Performance Impact of Plain Terms. We hypothesize that the proportion of plain terms to total terms should increase the variational solver's performance because as sharing grows, the query formula's variational core is reduced. We observe this behavior in Figure 8. Both $v \longrightarrow v$ and $p \longrightarrow p$ showed a statistically significant fit to a linear model. Furthermore, only $v \longrightarrow v$ was found to be statistically different from $p \longrightarrow p$ and $p \longrightarrow v$ with p-values of 4.67×10^{-4} and 1.10×10^{-4} thus confirming that sharing positively correlates to speedups for variational solving in these datasets.

This result is further evidence that as the reduction engine reduces more of the query formula, more reuse occurs, such as observed in the *auto* dataset where the sharing ratio is high. Hence,

[7]Using v0.2.5 of the gauge [55] library and v5.7.1 of the z3 [26] SMT solver with a solver seed set to 1729. All data was collected on a server running CentOS Linux release 7.7, with two Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz, 512GB RAM. We used stack lts-15.7 (GHC 8.8.3) and tested with RTS options "-qg -A64m -AL128m -n8m".
[8]https://github.com/lambda-land/VSat-Papers/tree/master/SPLC2020

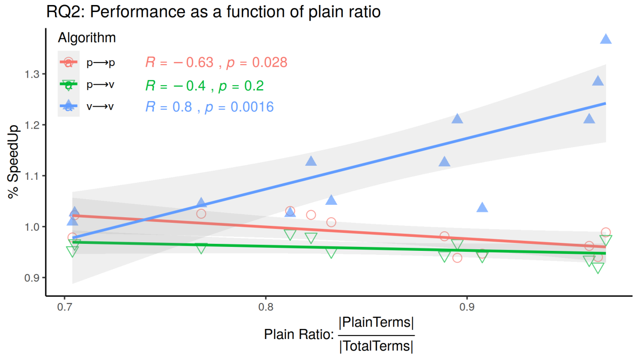


Figure 8: Sharing positively correlates to speedup only for $v \longrightarrow v$, where % SpeedUp $= \frac{\text{Algorithm}}{v \longrightarrow p}$.

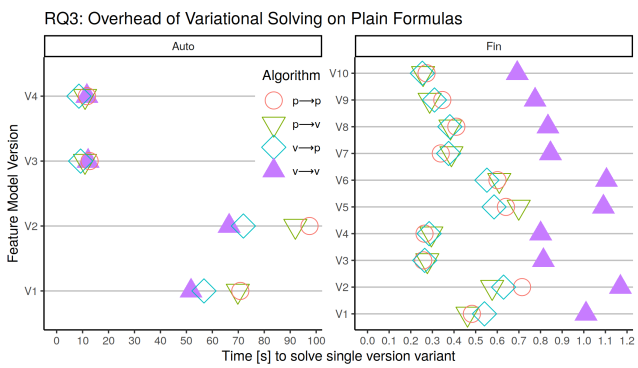


Figure 9: $v \longrightarrow v$ incurs an average slowdown of 17% for *auto*, and 60% for the *fin*, when solving a single {version}.

an avenue of future work is to leverage the laws of the variational logic to automatically refactor input formulas. The consequences of this observation will be particular to the application domain. For software product lines this means that any method to increase sharing between {versions} or SAT problems is desirable; this may be smaller changes with respect to the entire feature model, more frequent snapshots of the feature model, or syntactic manipulations to mitigate the occurrence of new features.

RQ3: Overhead of a Plain Query on VSAT. Figure 9 displays the bootstrapped averages of each version variant for each algorithm. We omit the matrix of pairwise comparisons from the paper for space, although it is available online. Of a total of 84 comparisons, 23 were significant in *fin* and 2 in *auto*. Given **RQ2**, and the composition of *fin*, we expect VSAT to show slowdowns for *fin*. This is observed in Figure 9 and is statistically significant for all versions. For *auto*, the only differences were in V_1, and between ($p \longrightarrow v$, $v \longrightarrow v$) and ($p \longrightarrow v$, $v \longrightarrow p$). Notably, $v \longrightarrow v$ did not differ from $v \longrightarrow p$, thus VSAT did not exhibit significant overhead for the *auto* dataset.[9]

[9] The *auto* portion of Figure 9 suggests statistically significant differences for other versions but omits variance, hence the discrepancy.

That $p \longrightarrow v$ was statistically different for V_1 suggests particular formulas may not respond well to the reduction engine. Similarly, there is clearly overhead when solving plain formulas, although this overhead is particular to some formulas, suggesting certain formula characteristics may have a large effect. Identifying these characteristics requires a more robust dataset; that some variants show no overhead suggests future work to recover performance.

Dataset	$v \longrightarrow v$	$v \longrightarrow p$	$p \longrightarrow v$	$p \longrightarrow p$
auto	211.70	288.66	363.16	378.69
fin	11.1	8.42	8.07	9.51

Table 1: Time to solve[s] Dead Core formula, $v \longrightarrow v$ shows a 76% speedup for *auto* data, and a 36% slowdown for *fin*.

Variational Dead and Core Analysis. Table 1 displays the performance results for the dead and core analyses. We observe a 76% speedup for the *auto* dataset, and a 36% slowdown for *fin* dataset. This difference is due to the threshold at which VSAT begins to outperform other algorithms. For *auto* this threshold was low, at 4 variants, but was 64 variants for the *fin* dataset, thus the slowdown. Following **RQ1**'s results, had a core, dead, *and* void analysis been performed, $v \longrightarrow v$ would still be under the speedup threshold.

Threats to Validity. Our results are subject to several threats to validity. Notably, we are unable to make absolute performance claims because our study, with only two product lines, may not be representative. To mitigate this we reused real-world data from Nieke et al.'s previous study [53] and chose dissimilar product lines. We inherit encoding-based threats to validity by reusing Nieke et al.'s formulas but ensured each algorithm experienced identical ordering of plain terms as described in Section 5.1.

Besides choice of dataset, our conclusions in the quantitative analysis are only representative of the performance of the z3 [26] SAT and SMT solver. While VSAT supports any SMTLIB2 [8] compliant solver, our evaluation used only z3. Due to z3's ubiquity we believe it to be representative of conflict-driven clause-learning SAT solvers, although other solvers could perform differently.

We have evinced the scalability claim with RQ1, and shown the translation and automation of incremental solving in Section 4. However, our results depend on a VPL formula as input. We believe that VPL formulas can be incrementally and automatically constructed in practice, as described in Section 3, as new variants occur or become known. However, assessing the usability and algorithmic challenges of VPL construction is left to future work.

In this paper, we do not provide a proof of the soundness of our methods. We mitigate this threat in several ways: we performed property-based testing [22] on our prototype and verified that a satisfiable variant was found to be satisfiable across all algorithms. In addition, we define a property that ensures that for each plain model p, found with $p \longrightarrow v$, $v \longrightarrow p$, and $p \longrightarrow p$, an identical model p' was found by substituting p on the variational model returned from VSAT. We performed the property-based tests with 3000 generated VPL formulas, finding no counter-examples.

6 RELATED WORK

Similar Solvers, Related Techniques. Our work is most similar to Visser et al. [66], which also constructs a SAT solver that exploits shared terms and prevents redundant computation. However, the projects differ in important ways. Visser et al.'s solver is oriented for program analysis and does not use incremental SAT solving. Rather, it uses heuristics to find canonical forms of sliced programs, and caches solver results on these canonical forms in a key-value store [41]. In contrast, variational SAT solving is domain agnostic, solves SAT problems expressed in VPL, returns a variational model, and uses incremental SAT solving.

Variational SAT solving is the latest in a line of work that uses the choice calculus to investigate variation as a computational phenomena. The choice calculus has been successfully applied to diverse areas of computer science, such as databases [4, 5], graphics [28], data structures [30, 49, 61, 69], type systems [14, 15, 20, 21], error messages [17–20], and now satisfiability solving. Our use of choices is similar to the concept of *facets* [6] and *faceted execution* [7, 50, 58], which have been successfully applied to information-flow security and policy-agnostic programming.

Applications for Variational Solving. Software variability, as explored in this paper, is a natural application domain for our work. The variability of SPLs or configurable software is often reduced to propositional logic [9, 25, 48] for analysis purposes [11, 32, 64]. Many analyses have been implemented using SAT solving [64], including feature-model analysis [11, 32], parsing [36], dead-code analysis [62], code simplification [67], type checking [63], consistency checking [24], dataflow analysis [44], model checking [23], variability-aware execution [52], testing [16], product sampling [47, 65], product configuration [57], optimization of non-functional properties [60], and variant-preserving refactoring [31]. While each of these analyses gives rise to multiple SAT problems for even a single analysis run, the authors typically do not discuss how they are solved. We argue that many could benefit from variational solving.

More generally, any scenario that involves solving many related SAT problems, and where all of these problems are known or can be generated in advance, is a potential application for variational SAT solving. Such situations arise in program analysis [66], and especially in *speculative* program analyses that involve generating and exploring huge numbers of variations of a program, for example, as in counterfactual [17] and migrational [14, 15] typing. Furthermore, we believe that variational solving provides a basis for such speculative analyses on feature models.

Efficient Reasoning about Software Variability. Since SAT solving is so common in software variability applications, many strategies have been developed to reduce effort in this domain.

Similar to variational formulas, Nieke et al. [53] encode several versions of a feature model in a single formula. We reuse their benchmark as part of our evaluation as described in Section 5.1; a direct comparison with their approach is nuanced and discussed in Section 5.2. While their work focuses on feature-model analysis only, variational formulas and variational solving can be applied to many application areas.

In the context of family-based type checking [64], others have discussed merging multiple SAT problems into one. Most work in this area use a *local* approach where SAT problems are solved as they are encountered during typing; in contrast, *global* approaches collect SAT checks into a single problem that is solved at the end of the analysis. While the global approach improves efficiency by increasing reuse of learned clauses in the solver, it loses the ability to identify *which* variants contain type errors [3, 34]. Variational solving can achieve the reuse benefits of the global approach without sacrificing the precision of the local approach.

Since the size of SAT problems in software variability applications is often dominated by the feature model, researchers tried to reduce the size of satisfiability problems by delaying consideration of the feature model until after the analysis and only using it rule out false positives [13, 23, 44], a technique known as late feature-model consideration [64]. Bodden et al. [13] found that this technique increases the overall efficiency of static analysis [13], while Classen et al. [23] found that it actually decreases efficiency of family-based model checking. Variational solving is orthogonal to these approaches since the feature model can be excluded from a variational formula and then used later to rule out false positives.

Feature models can also be reduced in size to speed up analyses, for example, by slicing [1, 39] or decomposition [59]. It is largely unexplored how much such reductions can improve efficiency, but the analysis will still involve multiple similar SAT problems, which can benefit from variational solving.

A final approach is to avoid SAT problems by using modal implications graphs [40], which support faster reasoning. The idea is to encode as many software variability constraints as possible in such graphs, then use a SAT solver only for the remaining constraints. The construction of modal implication graphs already requires solving SAT problems, but this cost is amortized if many SAT queries will be solved during the analysis, as Krieter et al. [40] found for configuration processes.

7 CONCLUSION

Variational satisfiability solving offers numerous advantages over current methods. Variational models, as solutions to variational satisfiability problems, are a flexible, compressed representation that enables post-hoc analyses. Through the use of a VPL formula, variational solving provides a domain agnostic, automated approach to use an incremental solver to efficiently solve sets of SAT problems, in addition to making explicit the ordering between plain and variational terms. Furthermore, we have demonstrated that sharing is an important factor in variational satisfiability solving. While the magnitude of its effect is not yet known, our analysis forms a foundation for future research. For feature modelers, variational satisfiability solving offers the practical benefits of a faster and more flexible analysis tool, and provides a basis for new kinds of automated variational analyses on feature models and software product lines. Outside the domain of software product lines, variational satisfiability solving provides a framework and logic where variation can be directly represented irrespective of the application domain, thus providing a new method to study variation itself.

ACKNOWLEDGMENTS

We gratefully acknowledge discussions with Michael Nieke and attendees of FOSD 2018. We thank colleagues without whom this work would not be possible: Paul Maximilian Bittner, Parisa Ataei, David Thrane Christiansen, and Levent Erkok. This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1).

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 424–427.

[2] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit Constraints in Partial Feature Models. In *Int. Work. on Feature-Oriented Software Development (FOSD)*. ACM, 18–27.

[3] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. 2010. Language-Independent Reference Checking in Software Product Lines. In *Int. Work. on Feature-Oriented Software Development (FOSD)*. ACM, 65–71.

[4] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. 2017. Variational Databases. In *Int. Symp. on Database Programming Languages (DBPL)*. ACM, 11:1–11:4.

[5] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. 2018. Managing Structurally Heterogeneous Databases in Software Product Lines. In *VLDB Workshop: Polystores and Other Systems for Heterogeneous Data (Poly)*.

[6] Thomas H Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 165–178.

[7] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS âĂŹ13)*. Association for Computing Machinery, New York, NY, USA, 15âĂŞ26. https://doi.org/10.1145/2465106.2465121

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`. (2016).

[9] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20.

[10] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. 2005. Automated Reasoning on Feature Models. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*. 491–503.

[11] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.

[12] Paul Maximilian Bittner, Thomas Thüm, and Ina Schaefer. 2019. SAT Encodings of the At-Most-k Constraint. In *Software Engineering and Formal Methods*, Peter Csaba Ölveczky and Gwen Salaün (Eds.). Springer International Publishing, Cham, 127–144.

[13] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 355–364.

[14] John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *Proc. of the ACM on Programming Languages (PACMPL)* issue *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)* 2 (2018), 15:1–15:29.

[15] John Peter Campora III, Sheng Chen, and Eric Walkingshaw. 2018. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. of the ACM on Programming Languages (PACMPL)* issue *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)* 2 (2018), 98:1–98:30.

[16] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)* 56, 10 (2014), 1183–1199.

[17] S. Chen and M. Erwig. 2014. Counter-Factual Typing for Debugging Type Errors. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. 583–594.

[18] S. Chen, M. Erwig, and K. Smeltzer. 2014. Let's Hear Both Sides: On Combining Type-Error Reporting Tools. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*. 145–152.

[19] S. Chen, M. Erwig, and K. Smeltzer. 2017. Exploiting Diversity in Type Checkers for Better Error Messages. *Journal of Visual Languages and Computing* 39 (2017), 10–21.

[20] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2012. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*. 29–40.

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

[22] Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 46, 4 (May 2011), 53âĂŞ64. https://doi.org/10.1145/1988042.1988046

[23] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. on Software Engineering* 39, 8 (2013), 1069–1089.

[24] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*. ACM, 211–220.

[25] 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.

[26] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[27] Jean Dunn and Olive Jean Dunn. 1961. Multiple Comparisons Among Means. *American Statistical Association* (1961), 52–64.

[28] M. Erwig and K. Smeltzer. 2018. Variational Pictures. In *Int. Conf. on the Theory and Application of Diagrams (LNAI 10871)*. 55–70.

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

[30] Martin Erwig, Eric Walkingshaw, and Sheng Chen. 2013. An Abstract Representation of Variational Graphs. In *Int. Work. on Feature-Oriented Software Development (FOSD)*. ACM, 25–32.

[31] 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.

[32] JosÃľ A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel GutiÃľrrez-Fernández, and Antonio Ruiz-CortÃľs. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (2019), 387–433.

[33] James Garson. 2018. Modal Logic. In *The Stanford Encyclopedia of Philosophy* (fall 2018 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[34] Shan Shan Huang, David Zook, and Yannis Smaragdakis. 2011. Statically Safe Program Generation with SafeGen. *Science of Computer Programming (SCP)* 76, 5 (2011), 376–391.

[35] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. Association for Computing Machinery, New York, NY, USA, 12âĂŞ1âĂŞ12âĂŞ55. https://doi.org/10.1145/1238844.1238856

[36] 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.

[37] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[38] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 132–143.

[39] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing Algorithms for Efficient Feature-Model Slicing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 60–64.

[40] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 898–909.

[41] Redis Labs. [n. d.]. Redis. https://redis.io/. ([n. d.]). Accessed at May 4th, 2020.

[42] Micahel Larabel. [n. d.]. A Global Switch To Kill Linux's CPU Spectre/Meltdown Workarounds? https://www.phoronix.com/scan.php?page=news_item&px=Global-Switch-Skip-Spectre-Melt. ([n. d.]). Accessed at March 25th, 2020.

[43] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Int. Conf. on Aspect-Oriented Software Development*. 191–202.

[44] 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.

[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[46] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2017. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 18–21.

[47] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 643–654.

[48] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient Compilation Techniques for Large Scale Feature Models. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*. ACM, 13–22.

[49] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. 2017. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 28–35.

[50] Kristopher K. Micinski. 2018. Abstracting Faceted Execution.

[51] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. 2014. Ultimately Incremental SAT. In *Theory and Applications of Satisfiability Testing – SAT 2014*, Carsten Sinz and Uwe Egly (Eds.). Springer International Publishing, Cham, 206–218.

[52] 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.

[53] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*. ACM, 188–201.

[54] National Institute of Standards and Technology. [n. d.]. NIST e-Handbook of Statistical Methods. https://www.itl.nist.gov/div898/handbook/index.htm. ([n. d.]). Accessed at May 7th, 2020.

[55] Bryan O'Sullivan. 2009. Criterian: A Haskell microbenchmarking library. Website. (2009). Available online at https://hackage.haskell.org/package/gauge-0.2.5; visited on May 7th, 2020.

[56] N. Rescher. 1969. *Many-valued Logic*. McGraw-Hill. https://books.google.com/books?id=ZyTXAAAAMAAJ

[57] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 465–474.

[58] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. 2018. Faceted Secure Multi Execution. In *CCS '18*.

[59] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678.

[60] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal (SQJ)* 20, 3-4 (2012), 487–517.

[61] K. Smeltzer and M. Erwig. 2017. Variational Lists: Comparisons and Design Guidelines. In *ACM SIGPLAN Int. Workshop on Feature-Oriented Software Development*. 31–40.

[62] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys)*. ACM, 47–60.

[63] Sahil Thaker, Don Batory, David Kitchin, and William Cook. 2007. Safe Composition of Product Lines. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*. ACM, 95–104.

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

[65] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–13.

[66] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*. ACM, 58:1–58:11.

[67] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 178–188.

[68] Eric Walkingshaw. 2013. *The Choice Calculus: A Formal Language of Variation*. Ph.D. Dissertation. Oregon State University. http://hdl.handle.net/1957/40652.

[69] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*. 213–226.

[70] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968