Noname manuscript No.
(will be inserted by the editor)

Variational Satisfiability Solving: Efficiently Solving Lots of Related SAT Problems

Jeffrey M. Young · Paul Maximilian Bittner · Eric Walkingshaw · Thomas Thüm

Received: date / Accepted: date

Abstract Incremental satisfiability (SAT) solving is an extension of classic SAT solving that enables solving a set of related SAT problems by identifying and exploiting shared terms. However, using incremental solvers effectively is hard since performance is sensitive to the input order of subterms and results must be tracked manually. For analyses that generate sets of related SAT problems, such as those in software product lines, incremental solvers are either not used or their use is not clearly described in the literature. This paper translates the ordering problem to an encoding problem and automates the use of incremental solving. We introduce *variational* SAT solving, which differs from incremental solving by accepting all related problems as a single variational input and returning all results as a single variational output. Variational solving syntactically encodes differences in related SAT problems as local points of variation. With this syntax, our approach automates the interaction with the incremental solver and enables a method to automatically optimize sharing in the input. To evaluate these ideas, we formalize a variational SAT algorithm, construct a prototype variational 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

Statements and Declarations

This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1).

Jeffrey M. Young
IOHK, Longmont, Colorado, E-mail: jeffrey.young@iohk.io

Paul Maximilian Bittner
University of Ulm, Ulm, Germany, E-mail: paul.bittner@uni-ulm.de

Eric Walkingshaw
Unaffiliated, Corvallis, OR, USA, E-mail: ewalkingshaw@acm.org

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

the prototype solver scales better for these problems than four off-the-shelf incremental solvers while also automatically tracking individual results.

Keywords satisfiability solving · variation · choice calculus · software product lines

1 Introduction

Satisfiability (SAT) solving is a ubiquitous technology in software product lines for a diverse set of analyses ranging from anomaly detection (Mauro et al., 2017; Kowal et al., 2016; Ananieva et al., 2016), dead code analysis (Tartler et al., 2011), sampling (Medeiros et al., 2016; Varshosaz et al., 2018), and automated analysis of feature models (Benavides et al., 2005; Thüm et al., 2014; Galindo et al., 2019). The general pattern is to represent parts of the system or feature model as a propositional formula (Batory, 2005; Czarnecki and Wąsowski, 2007; Mendonça et al., 2008), and reduce the analysis to a 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 FM_i for some product version i, represented as a conjunction of clauses that describe the relationships among features: $FM_i = c_0 \wedge c_1 \wedge \dots \wedge c_n$. Now consider a case where several properties must be guaranteed for every commit via a continuous integration tool (e.g., via a void feature model analysis or dead feature analysis). One might add new clauses, remove clauses, or alter clauses depending on the property. For example:

$$\begin{aligned}
SAT_{FM_i\_\text{void}} &= c_0 \wedge c_1 \wedge c_2 \wedge c_3 \wedge \dots \wedge c_n \\
SAT_{FM_i\_\text{dead}} &= (c_0 \wedge c_1 \wedge c_2 \wedge c_3 \wedge \dots \wedge c_n) \wedge \neg \mathit{core\_feature} \\
&\vdots \\
SAT_{FM_{i+n}\_\text{dead}} &= (c_0' \wedge c_1 \wedge c_2 \wedge c_3 \wedge \dots \wedge c_n') \wedge \neg \mathit{other\_core\_feature}
\end{aligned}$$

Where a tick mark $'$, indicates clauses which have been altered from version FM_i. Or consider a case where one might perform a single analysis over several versions or commits yielding a set of SAT problems

$$\begin{aligned}
SAT_{FM_i} &= (c_0 \wedge c_1 \wedge c_2 \wedge c_3 \wedge \dots \wedge c_n) \wedge \mathit{dead\_feature} \\
SAT_{FM_{i+1}} &= (c_0 \wedge c_1' \wedge c_2 \wedge c_3' \wedge \dots \wedge c_n) \wedge \mathit{dead\_feature} \\
&\vdots \\
SAT_{FM_{i+n}} &= (c_0' \wedge c_1' \wedge c_2 \wedge c_3'' \wedge \dots \wedge c_n') \wedge \mathit{dead\_feature}
\end{aligned}$$

Where a double tick mark, $''$, indicates a clause that has been altered more than once from version FM_i. In such cases, state-of-the-art methods do not make use of commonalities among the set of formulas and perform many redundant computations.

A concrete example of the above scenario naturally occurred in the Linux Foundation's response to the meltdown and spectre security vulnerabilities (Kocher et al., 2019; Lipp et al., 2018). 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 leads to a set of SAT problems, where each problem represents verification of some property of the feature model, or the same analysis over many versions of a feature model.

In these cases, analysis over the set of SAT problems produced by the kernel changes leads to a dilemma. We must either perform the analysis on each SAT problem, and thus feature model, individually, therefore not making any use of previously known commonalities. Or we might try to reuse results from previous SAT calls 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 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. Theoretically, this is more efficient because it reuses knowledge of shared terms, however, using an incremental solver in this way requires substantial manual effort and domain knowledge, produces a specific solution to a specific analysis, and it requires extra infrastructure to manage results. Hence, we are left with either an inefficient analysis full of redundancies, or with an efficient analysis that requires a high degree of manual effort and domain knowledge.

Many analyses and tools have been developed to deal with code-level variability in software, which make use of incremental solvers to solve lots of related SAT problems, as described above. For example, Kästner et al. (2011) constructed a variation-aware parser (Kästner et al., 2011), lexer (Kästner et al., 2011), and type checker (Kästner et al., 2012; Liebig et al., 2013) to type check all possible configurations of the Linux kernel. Additionally, Meinicke (2014, 2019) constructed a variation-aware bytecode, which supports debugging software product lines, performing mutation analysis, and automated program repair Wong (2021). Finally, Ataei et al. (2021a) created a variational database management system and query language that enables working with variation in the structure or content of data arising for a wide variety of reasons, such as due to schema evolution, data integration, or integration with a software product line Ataei et al. (2018). Each of these systems must efficiently reason about sets of related domain artifacts encoded as SAT problems. Incremental solvers fill this need, but again, require significant effort and domain knowledge to use effectively.

In this paper, we show that the performance benefit of using an incremental solver to solve a large number of related SAT problems can be achieved while mitigating the main drawbacks of incremental solvers, that is, high manual effort and deep integration with the application. Our solution is to formalize a method of *variational* SAT solving that makes use of known commonalities among propositional formulas and automates the interaction with an incremental solver. Variational SAT solving takes as input a single *variational formula*, which encodes a set of related SAT problems to solve. It then compiles this variational formula into an efficient program to run on an incremental solver. Finally, it collects the results from the incremental solver into a single result that captures the solutions to all of the SAT problems described by the input variational formula. We call the input formula a *variational* formula since it efficiently encodes many formulas in one by differentiating parts of the formula that are shared among many formulas, and thus that *vary* among some of the formulas. A variational SAT solver then solves variational formulas.

Our approach has several 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, which can be constructed from any set of related SAT problems. (3) With a variational formula, new kinds of syntactic manipulations and optimizations, 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 SAT solving and assume a variational formula as input, leaving other considerations such as the optimal encoding of such formulas to future work. To summarize, we make the following contributions:

– 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 4.1)
– We present and formalize an algorithm that solves formulas in the extended logic using off-the-shelf incremental solvers as black boxes.

– We construct, evaluate, and compare a prototype variational satisfiability solver, on two real-world datasets, to four incremental satisfiability solvers. The prototype variational satisfiability solver is publicly available.[1] (Section 4.2)
– We report reduced execution times for the prototype solver compared to all tested incremental solvers when solving many variants. All analysis scripts and data are publicly available.[2](Section 5.2)

This work builds on the conference version (Young et al., 2020) of this paper in two ways. First, we present the inference rules that formalize the behavior of a variational SAT solver. Second, we repeat the conference version's experiment with three more incremental solvers. The conference paper evaluation used only the Z3 (de Moura and Bjørner, 2008) solver as the variational solver's backend, which we highlighted as a threat to validity. In this version, we address that threat by repeating our analysis with CVC4 (Barrett et al., 2011), Boolector (Brummayer and Biere, 2009), and Yices (Dutertre, 2014).

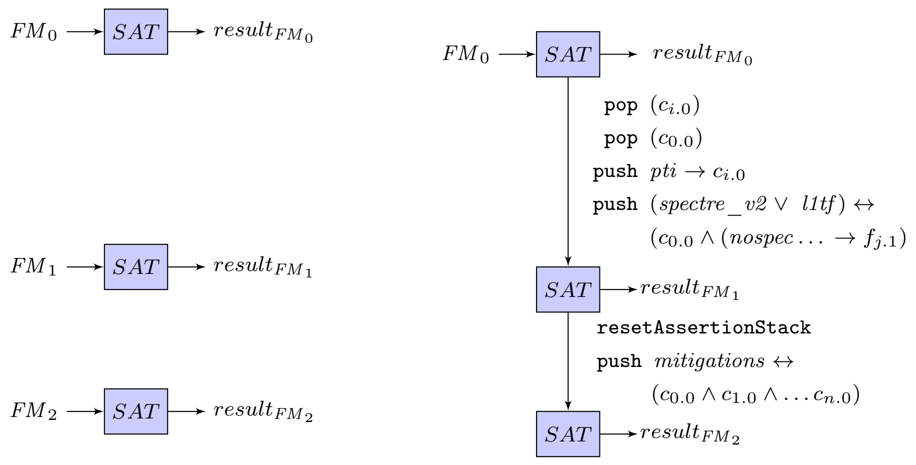
2 Background

Variational SAT solving depends on incremental SAT/SMT solving. In this section, we describe the underlying data structures and operations that variational SAT solving exploits, using the Linux Kernel as a running example. Our description, and the interface between variational SAT solving and incremental SAT/SMT conforms to the SMTLIB2 (Barrett et al., 2016) 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.[3]

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 that originated in the jth feature model. For formulas, we use $c_{i.j}$ to refer to the formula that originated in the jth feature model and that encodes the ith feature's relationships to other features. Thus, the feature models can

[1] `https://doi.org/10.5281/zenodo.5543884`
[2] `https://doi.org/10.5281/zenodo.5546009`
[3] The feature names are from the Linux kernel, see Larabel (2020).



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

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

Fig. 1: Conceptual difference between incremental and brute force SAT procedures.

be represented by the following formulas:

$$\begin{aligned} FM_0 &= c_{0.0} \land c_{1.0} \land \ldots \land c_{n.0} \\ FM_1 &= (spectre\_v2 \lor l1tf) \leftrightarrow (c'_{0.0} \land (nospec\_store\_bypass\text{-}disable \\ &\quad \to f_{j.1}) \land c_{1.0} \land (pti \to c'_{i.0}) \land \ldots \land c_{n.0}) \\ FM_2 &= mitigations \leftrightarrow (c_{0.0} \land c_{1.0} \land \ldots \land c_{n.0}) \end{aligned}$$

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: $(pti \to c_{i.0})$, $(spectre\_v2 \lor l1tf)$, $(nospec\_store\_bypass\text{-}disable \to f_{j.1})$, 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 (called a *model*, not to be confused with a feature model) for each formula. Using a classic SAT solver results in the procedure illustrated in Figure 1a, where the SAT solver is applied to each formula separately; the solver does not recognize information shared between the formulas and does not propagate information learned during the solving procedure to future solving procedures. Alternatively, using an *incremental* solver is illustrated in Figure 1b; in this scenario, all of the formulas are solved by a single solver instance where terms are programmatically added and removed from the solver throughout the process. The ability to add and remove terms from the solvers is enabled by a data structure within the incremental 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 (Nadel et al., 2014).

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 subformulas 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 (Erwig and Walkingshaw, 2011; Walkingshaw, 2013), 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* 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 (Liebig et al., 2011) application of `#ifdef` annotations. From a logical perspective, following the many-valued logic of Kleene (Kleene, 1968; Rescher, 1969), 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

$$
\begin{array}{lll}
t ::= r \quad | \quad \mathtt{T} \quad | \quad \mathtt{F} & & \textit{Variables and Boolean literals} \\[6pt]
f ::= t & & \textit{Terminal} \\
\quad | \ \neg f & & \textit{Negate} \\
\quad | \ f \vee f & & \textit{Or} \\
\quad | \ f \wedge f & & \textit{And} \\
\quad | \ D\langle f, f\rangle & & \textit{Choice}
\end{array}
$$

(a) Syntax of VPL.

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

(b) Configuration semantics of VPL.

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

(c) VPL equivalence laws.

Fig. 2: Formal definition of VPL.

as modal logic (Garson, 2018), 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 are 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 $C(A) = \mathsf{true}$ and $p \wedge r$ when $C(A) = \mathsf{false}$. 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[4] 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.

[4] We hypothesize that it is equivalent to BDD minimization, which is NP-complete, but the equivalence has not been proved; see Walkingshaw et al. (2014).

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 because the syntactic distance between the two is 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.

$$f_{FM_{02}} = L_2\langle \mathit{mitigations}, \mathtt{T}\rangle \leftrightarrow c_{0.0} \wedge c_{1.0} \wedge \ldots \wedge c_{n.0}$$

We exploit the fact that $\mathtt{T} \leftrightarrow c_{0.0} \wedge \ldots$ is equivalent to $c_{0.0} \wedge \ldots$ to recover a formula equivalent to FM_0 for configurations where L_2 is false. [5]

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:

$$\begin{aligned} f_{FM_{012}} = {} & L_1\langle(\mathit{spectre\_v2} \vee \mathit{l1tf}), L_2\langle \mathit{mitigations}, \mathtt{T}\rangle\rangle \\ & \leftrightarrow L_1\langle(c_{0.0} \wedge (\mathit{nospec\_store\_bypass\text{-}disable} \rightarrow f_j), c_{0.0}\rangle \\ & \wedge L_1\langle c_{1.0}, \mathtt{T}\rangle \wedge c_1 \wedge L_1\langle(\mathit{pti} \rightarrow c_{i.0}), \mathtt{T}\rangle \wedge \ldots \wedge c_n \end{aligned}$$

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 present our algorithm for variational satisfiability solving. Section 4.1 provides an overview of the algorithm and introduces the notion of *variational models* as solutions to variational SAT problems. Section 4.2 provides the formal specification.

4.1 General Approach

We solve VPL formulas recursively, decoupling the handling of plain terms from the handling of variational terms. The intuition behind our algorithm is

[5] Consider $\mathtt{T} \leftrightarrow p$, this is equivalent to $(\mathtt{T} \rightarrow p) \wedge (p \rightarrow \mathtt{T})$. Notice that for $p \rightarrow \mathtt{T}$ the truth value of p does not matter because $\mathtt{F} \rightarrow \mathtt{T} \equiv \mathtt{T}$ and $\mathtt{T} \rightarrow \mathtt{T} \equiv \mathtt{T}$. This leaves just $\mathtt{T} \rightarrow p$, which is $\mathtt{T}$ if and only if $p \equiv \mathtt{T}$, and thus $\mathtt{T} \rightarrow p \equiv p$. Finally, we get $\mathtt{T} \leftrightarrow p \equiv p$.

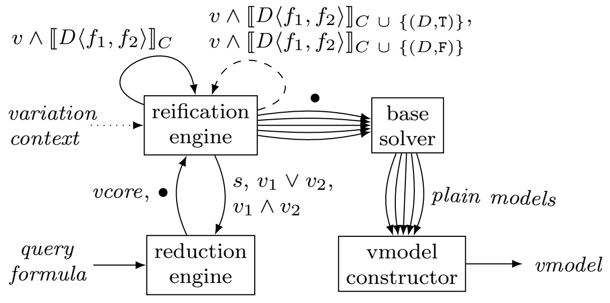

$v \wedge [\![D\langle f_1, f_2\rangle]\!]_C$
$v \wedge [\![D\langle f_1, f_2\rangle]\!]_{C\,\cup\,\{(D,T)\}}$,
$v \wedge [\![D\langle f_1, f_2\rangle]\!]_{C\,\cup\,\{(D,F)\}}$
variation context
reification engine
●
base solver
vcore, ●
s, $v_1 \vee v_2$, $v_1 \wedge v_2$
plain models
query formula
reduction engine
vmodel constructor
vmodel

Fig. 3: System overview of the variational solver.

to first process as many plain terms as possible (e.g., by pushing those terms to the underlying solver) while skipping choices, yielding a *variational core* that represents only the variational parts of the original formula. We then alternate between configuring choices in the variational core and processing the new plain terms produced by configuration until the entire term has been consumed. Each consumption of the core corresponds to one variant of the original VPL formula since all of its choices will have been configured in a particular way. At which point, we query the underlying solver to obtain a model for that variant, then backtrack to solve another variant by configuring the choices differently. Finally, the models for all variants are combined into a single *variational model* that captures the result of solving all variants of the original VPL formula.

We present a simplified overview of the variational solver as a state diagram in Figure 3 that operates on the input's abstract syntax tree. Labels on incoming edges denote inputs to a state, while 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 $\wedge$.

The variational solver has four subsystems:

- The *reduction engine* processes plain terms and generates the variational core, which is ready for reification.
- The *reification engine* configures choices in a variational core.
- The *base solver* is an off-the-shelf incremental solver (such as Z3 (de Moura and Bjørner, 2008)) or Boolector (Brummayer and Biere, 2009)), used to produce plain models.
- The *variational model constructor* synthesizes a single variational model from the set of plain models returned by the base solver.

The variational 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. For example, passing $vc = D_1 \wedge D_2$ would restrict the solver to consider only variants in which both D_1 and D_2 are T. The variational solver translates the query formula to a formula in an intermediate language (IL) that the reduction and reification engines operate over. The syntax of the IL is given below.

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

The IL includes two kinds of terms not present in the input query formulas: First, the $\bullet$ value, pronounced *unit*, represents terms that have already been fully processed and sent to the base solver. Recall that the base solver is incremental; during reduction, we will incrementally push subterms of the formula to the base solver's assertion stack (see Section 2) and replace those subterms with $\bullet$. Second, a *symbolic value s* represents a plain subterm that has been reduced but cannot yet be sent to the base solver (see Section 4.1). Also, note that choices contain VPL formulas (f) as alternatives, not IL formulas. Thus, an IL formula is a mix of partially processed subterms outside of choices and unprocessed formulas within choices.

Reduction Engine: Derivation of a Variational Core. Reduction transforms an IL formula into a variational core. A *variational core* consists of (1) a set of assertions that have been sent to the base solver, and (2) a *fully reduced* IL formula. A fully reduced IL formula consists only of unit values, $\bullet$; symbolic references, s; logical connectives and choices, with the invariant that all connectives must contain at least one choice. For example, the IL formula $s \wedge D\langle f_1, f_2\rangle$ is fully reduced, while $r \wedge D\langle f_1, f_2\rangle$ is not fully reduced since it contains a (non-symbolic) variable reference r. As another example, $(s_1 \wedge s_2) \vee D\langle f_1, f_2\rangle$ is not fully reduced since the subterm $s_1 \wedge s_2$ contains a logical connective without a choice as a descendant. Therefore, the subterm $s_1 \wedge s_2$ has to be reduced to another symbolic value s_3 first.

The goal of reduction is to *do as much work as possible without configuring any choices.* During variational solving, each state where we configure a choice is a state that we must backtrack to in the future (to select the other alternative), so we want to maximize the work we do before reaching each backtracking point. A variational core represents an intermediate state where we cannot make any more progress without configuring a choice.

The variational core for a VPL formula is computed by the reduction engine illustrated in Figure 4. The reduction engine has two states: *evaluation*, which communicates to the base solver to process plain terms, and *accumulation*, which is called by evaluation to create symbolic references.

To illustrate how the reduction engine computes a variational core, consider the query formula $f = ((a \wedge b) \wedge A\langle f_1, f_2\rangle) \wedge ((p \wedge \neg q) \vee B\langle f_3, f_4\rangle)$. Translated to an IL formula, f has four references (a, b, p, q) and two choices. Generating the core begins with evaluation. Evaluation matches the root $\wedge$ node of f and recurs following the $v_1 \wedge v_2$ edge, where $v_1 = (a \wedge b) \wedge A\langle f_1, f_2\rangle$ and $v_2 = (p \wedge \neg q) \vee B\langle f_3, f_4\rangle$.

When evaluating v_1, evaluation again matches on $\wedge$, creating another recursive call with $v_1' = a \wedge b$ and $v_2' = A\langle f_1, f_2\rangle$. One more match on $\wedge$ within v_1' yields $v_1'' = a$ and $v_2'' = b$. In this call, both a and b are references, so

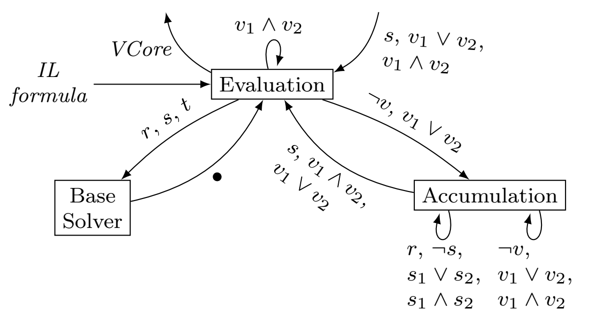

VCore
$v_1 \wedge v_2$
s, $v_1 \vee v_2$, $v_1 \wedge v_2$
IL formula
Evaluation
r, s, t
$\bullet$
s, $v_1 \wedge v_2$, $v_1 \vee v_2$
$\neg v$, $v_1 \vee v_2$
Base Solver
Accumulation
r, $\neg s$, $s_1 \vee s_2$, $s_1 \wedge s_2$
$\neg v$, $v_1 \vee v_2$, $v_1 \wedge v_2$

Fig. 4: Overview of the reduction engine.

evaluation sends each in turn to the base solver, following the r, s, t edge, yielding $\bullet$ for each. The resulting term $\bullet \wedge \bullet$ will be further reduced to simply $\bullet$, which is returned as the result of evaluating v_1'. This value indicates that the entire term v_1' has been send to the base solver. At this point, we have $v_1' = \bullet$ and $v_2' = A\langle f_1, f_2\rangle$. The term v_2' is a choice, which cannot be reduced during evaluation, so the result of evaluating v_1 is $\bullet \wedge A\langle f_1, f_2\rangle$.

The evaluation of v_1 demonstrates that evaluation can process conjunction nodes by simply sending plain subterms to the base solver and leaving choices alone (to be processed later). This works because the terms in the base solver's assertion stack are implicitly conjuncted together. When we process the choices later, we can simply push alternatives to the stack, and when we backtrack, we can pop them back off again and push different alternatives. In contrast, we cannot blithely push the children of a disjunction or negation node onto the assertion stack since the semantics of the assertion stack does not align with these logical operations. However, we still want to somehow process these nodes so that we do not have to perform redundant work each time we backtrack. Instead of evaluating them by sending them to the base solver, we instead *accumulate* them into symbolic values to be reused later. This process is illustrated during the evaluation of v_2.

When reducing the right child, $v_2 = (p \wedge \neg q) \vee B\langle f_3, f_4\rangle$ of f, evaluation will match on the root $\vee$ and transition via the edge labeled $v_1 \vee v_2$. This will split v_2 into $v_1' = p \wedge \neg q$ and $v_2' = B\langle f_3, f_4\rangle$. Note that the subterm v_1' is a plain expression. However unlike evaluation, this expression will not be sent to the base solver, but will instead be "accumulated" to a symbolic value. A symbolic value is a place holder for an arbitrarily large plain subterm. This enables us to process the subterm only once (during the accumulation phase), and then (re-)use the symbolic value throughout the rest of variational solving.

Accumulation of a plain subterm is a recursive function that traverses down the formula's syntax tree, eventually replacing variables at the leaves by symbol values, then accumulating each subterm that consists only of logical operations applied to symbolic values with a new symbolic value. A mapping (explained below) is maintained to enable replacing identical subterms with the same symbolic values to maximize sharing. In the case of v'_1, the variables p and q will be replaced by symbolic values s_p and s_q yielding the intermediate term $s_p \wedge \neg s_q$. Then $\neg s_q$ will be replaced by a new symbolic value s'_q yielding $s_p \wedge s'_q$. Finally, since this term consists of a logical operation applied to symbolic values, the entire subterm will be accumulated to a new symbolic value s_{pq}.

Since v'_2 is a choice, there is nothing left to do during accumulation, so the result of accumulating v_2 is $s_{pq} \vee B\langle f_3, f_4\rangle$, which is returned to evaluation. Finally, since both v_1 and v_2 have been fully reduced, we obtain the variational core $A\langle f_1, f_2\rangle \wedge (s_{pq} \vee B\langle f_3, f_4\rangle)$ for f.

The variational core represents an intermediate term where as much work has been done as possible—either by sending terms to the base solver or by accumulating subterms to simpler symbolic values—before we have to process the next choice. If instead we had simply solved f by recursively evaluating the subterms of f, plain subterms, such as $a \wedge b$ and $p \wedge \neg q$, would be processed once for each variant even though they are unchanged. Evaluation moves subterms into the solver state to be reused among different variants. Accumulation processes and caches subterms that cannot be immediately evaluated so that they can be efficiently evaluated later.

The symbolic values produced by accumulation correspond to variables in the reduction engine's memory that represent a set of statements *declared in* but not yet *sent* to the base solver.[6] For example, the symbolic value s_{pq} represents three declarations in the base solver:

```
(declare-const p Bool)
(declare-const q Bool)
(declare-fun s_pq () Bool (and p (not q)))
```

A variational core repesented in the base solver is a sequence of statements that contain "holes" ◊, which correspond to the choices that have yet to be processed. For example, the variational core of f could be encoded as:

```
(declare-const ◊)           ;; choice A
  ⋮                         ;; declarations and assertions for alternatives
(declare-const ◊)           ;; choice B
  ⋮                         ;; declarations and assertions for alternatives
(assert (or s_pq ◊))        ;; assertion waiting on the configuration of B
```

This is not yet a program that can be executed by the base solver, but rather a template that will be filled in when we configure a choice, *then* sent to the base solver. The constant declared for each choice tracks how we configured the choice, while the ⋮ will be filled with declarations and assertions corresponding to the configured alternative. Since we are sending this template to the base solver potentially many times, it is important that it is as small as possible. We achieved this in our example through reduction by asserting $a \wedge b$ once during evaluation (so that it doesn't appear in the variational core at all) and

[6] In this section, we use SMTLIB2 snippets to represent operations performed on the base solver. While we target SMTLIB2, conforming to the standard is not a requirement. Any solver, such as Minisat (Eén and Sörensson, 2004), that exposes an incremental API, such as IPASIR (SAT Competition, 2021), can be used to implement variational SAT solving.

accumulating $p \wedge \neg q$ to s_{pq} once so that we can reuse it in each use of the template.

To summarize: The reduction engine produces a variational core by recursively processing an IL term. Plain subterms outside of choices are either sent to the base solver (in which case they do not appear in the variational core) or are cached and replaced in the variational core by a symbolic value. Reduction does not change any subterms within choices. Therefore, a variational core produced by reduction consists only of top-level logical connectives applied to choices and/or symbolic values.

Reification Engine: Solving the Variational Core. Once a variational core is obtained from the reduction engine, it is sent to the reification engine to be solved. The reification engine processes a variational core by configuring its choices and solving each variant. Reification ensures that eventually all variants of the variational core will be explored by implementing a backtracking algorithm.

In each configuration step, the reification engine produces a new IL term in which all choices in one dimension have been eliminated. This new term is immediately sent back to the reduction engine to obtain a new variational core that will be sent back to the reification engine, and so on. In this way, the solver alternates between reduction and reification until all choices have been eliminated. At which point, a solution in the form of a satisfiable assignment is obtained for that variant from the base solver (see the discussion of *variational models*, later in this section) and the reification engine backtracks to solve the next variant.

The reification engine keeps track of which alternative is selected from each choice by maintaining a *configuration* value C. In Section 3, we formalized a configuration as a function $D \rightarrow \mathbb{B}_{\perp}$ (i.e., a function from dimension names to the domain of Boolean values which includes a single extra element; $\perp$).

A dimension mapped to $\mathtt{T}$ indicates that the left alternatives of all choices in that dimension have been selected, while $\mathtt{F}$ indicates the right alternatives have been selected. A dimension mapped to $\perp$ has not yet been configured. In this section, we represent configurations extensionally as sets of tuples $D \times \mathbb{B}$. For example, the configuration $C = \{(A, \mathtt{T}), (B, \mathtt{F})\}$ maps dimension A to $\mathtt{T}$, dimension B to $\mathtt{F}$, and all other dimensions to $\perp$. The configuration value is initialized to the variation context passed into the solver and is updated during reification.

When a choice is encountered during reification, we check to see whether its dimension is defined in the configuration, that is we check $C(D) \neq \perp$. If $C(D) \neq \perp$, then we replace the choice by the corresponding alternative. For example, given a variational core $s \wedge A\langle f_1, f_2\rangle$, if $(A, \mathtt{T}) \in C$, then the formula is reified to $s \wedge f_1$, which is sent to the reduction engine; if $(A, \mathtt{F}) \in C$, then $s \wedge f_2$ is sent to the reduction engine. If the dimension is not defined in the configuration, then the reification must explore *both* possibilities. It does this by first extending the configuration to explore the left branch. When that path has been fully processed, it backtracks to extend the configuration to explore the right branch. For example, if $C(A) = \bot$, then reification explores the left alternative with $C' = C \cup \{(A, \mathtt{T})\}$, finishes, then backtracks to explore the right alternative with $C' = C \cup \{(A, \mathtt{F})\}$.

To the base solver, the backtracking performed by reification is implemented as a linear sequence of assertion stack manipulations. For example, reifying $s \wedge A\langle f_1, f_2\rangle$ when $C(A) = \bot$ corresponds to a sequence of solver commands such as the following:

```
  ⋮            ;; declarations and assertions for the variational core
(push 1)       ;; create a new context for the left alternative
  ⋮            ;; declarations defining the left alternative to be $s_{f_1}$
(declare-fun $s_{A_T}$ () Bool (and $s$ $s_{f_1}$)) ;; define formula containing alternative
(assert $s_{A_T}$)
  ⋮            ;; solve this formula
(pop 1)        ;; done with the left alternative
(push 1)       ;; create a new context for the right alternative
  ⋮            ;; declarations defining the left alternative to be $s_{f_1}$
(declare-fun $s_{A_F}$ () Bool (and $s$ $s_{f_2}$)) ;; define formula containing alternative
(assert $s_{A_F}$)
  ⋮            ;; solve this formula
(pop 1)        ;; done with the right alternative
```

The backtracking is realized through the use of `push` and `pop` to create a new context to solve each alternative, and to restore the original state after we're done. The `1` arguments to `push` and `pop` indicate that we're pushing/popping one level of context to the assertion stack. The lines beginning with `declare -fun` define the formula to be solved after configuring the choice; it uses the left alternative or right alternative by substituting the alternative into the variational core where the choice used to be.

Variational Models. The solution to a plain SAT problem, called a *model*, is a Boolean value indicating whether the formula is satisfiable, and if so, an assignment of T or F to all of the Boolean variables in the formula such that the formula evaluates to T. We define a *variational model* as a data structure that enables recovering a plain model for each variant of a variational formula. The goal of a variational model is to compactly and accurately represent the plain models produced by the base solver for each satisfiable variant.

While plain models map variables to Boolean values, variational models map variables to variation contexts (Boolean formulas of dimensions) that record in which variants the variable was assigned T. We denote the variation context for a variable r as vc_r. For example, the mapping $vc_r = A \wedge B$ indicates that variable r was assigned T in models where both dimensions A and B have been configured to T, and r was assigned F otherwise.

Additionally, variational models maintain a special variation context mapped to a variable called *Sat*, which tracks the configurations found to be satisfiable. With *Sat* we can use variational models to find a single satisfiable configuration by applying a plain SAT solver to the *Sat* variation context, or we can enumerate all satisfiable configurations by applying the all-SAT operation supported by most SAT solvers, which would return all models that make *Sat* satisfiable.

$a \to \mathtt{T}$	$a \to \mathtt{T}$	$a \to \mathtt{T}$
$b \to \mathtt{F}$	$b \to \mathtt{F}$	$b \to \mathtt{F}$
	$c \to \mathtt{T}$	$c \to \mathtt{T}$
$p \to \mathtt{F}$	$p \to \mathtt{T}$	$p \to \mathtt{T}$
$q \to \mathtt{T}$	$q \to \mathtt{F}$	$q \to \mathtt{F}$
$C_{TT} = \{(A, \mathtt{T}), (B, \mathtt{T})\}$	$C_{FT} = \{(A, \mathtt{F}), (B, \mathtt{T})\}$	$C_{FF} = \{(A, \mathtt{F}), (B, \mathtt{F})\}$

Fig. 5: Possible plain models for variants of f.

$$\begin{aligned}
&\mathit{Sat} \to && (\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B) \\
&a \to && (\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B) \\
&b \to && \mathtt{F} \\
&c \to && (\neg A \wedge \neg B) \vee (\neg A \wedge B) \\
&p \to && (\neg A \wedge \neg B) \vee (\neg A \wedge B) \\
&q \to && (A \wedge B)
\end{aligned}$$

Fig. 6: Variational model of the plain models in Figure 5.

As an example, consider the query formula f:

$$f = ((a \wedge \neg b) \wedge A\langle a \to \neg p, c\rangle) \wedge ((p \wedge \neg q) \vee B\langle q, p\rangle)$$

We see that the variant, with configuration $\{(A, \mathtt{T}), (B, \mathtt{F})\}$ is unsatisfiable, because a must always be $\mathtt{T}$, which forces p be $\mathtt{F}$, which means $(p \wedge \neg q) \vee p$ is always $\mathtt{F}$. Assuming each other variant is satisfiable, there are a minimum of three possible plain models. Figure 5 shows these example plain models while the corresponding variational model is shown in Figure 6. The variation context *Sat* consists of three disjuncted terms, one for each satisfiable variant (given by C_{TT}, C_{FT}, and C_{FF}). We can retrieve the value of any variable x in any configuration by substituting the configuration's values in the variables variation context vc_x. For example, we can retrieve a's value in the plain model returned for $C_{FT} = \{(A, \mathtt{F}), (B, \mathtt{T})\}$ in the following way:

$$\begin{aligned}
a &\equiv (\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B) && vc \text{ for } a \\
&\equiv (\neg \mathtt{F} \wedge \neg \mathtt{T}) \vee (\neg \mathtt{F} \wedge \mathtt{T}) \vee (\mathtt{F} \wedge \mathtt{T}) && \text{Substitute } \mathtt{F} \text{ for } A, \mathtt{T} \text{ for } B \\
&\equiv \mathtt{T} && \text{Result}
\end{aligned}$$

Additionally, we can compute all variants where a variable j is satisfiable by solving for all possible models of vc_j with all-SAT(vc_j).

Variational models are constructed incrementally by merging each new plain model returned by the base solver into the variational model. A merge requires the current configuration, the plain model, and the current vc of the variant being solved. The variational model is empty until the first plain model is emitted from the base solver. Once a plain model is emitted from the base solver, the variational model constructor iterates over each variable in the variational model updating that variables assignment in the variational model. There are four cases:

1. If a variable is assigned $\mathtt{F}$ in the plain model and is assigned $\mathtt{F}$ in the variational model, then it remains assigned to $\mathtt{F}$; b in Figure 5 and Figure 6

is one such example. We call variables that are never assigned a vc, such as b, *constant* because they are independent of the variants.

2. If a variable is assigned `F` in the plain model, but is assigned a vc in the variational model, then it is skipped and its' assignment in the variational model is not updated. For example, if the plain models shown in Figure 5 were emitted in order from C_{TT} to C_{FF} then q's assignment is not updated when the C_{FT} and C_{FF} plain models are merged, resulting in q's final assignment shown in Figure 6.
3. If a variable is assigned `T` in the plain model but `F` in the variational model then its' entry in the variational model must be updated. The vc of that variable is set to the formula that describes the satisfiable variant. For example, if the plain models shown in Figure 5 were emitted in order from C_{TT} to C_{FF} then the assignment of p in the variational model changes from `F` once the C_{FT} plain model is merged.
4. If a variable is assigned `T` in the plain model, and is assigned a vc in the variational model, then a new vc is constructed by pre-pending the current vc to the assigned vc with $\vee$. For example, vc_a in Figure 6 shows the order of merges: The oldest vc is last in the chain of $\vee$'s (i.e., the one representing C_{TT}), while the most recent is the vc representing C_{FF}; the last model to be emitted and merged from the base solver.

To summarize: Variational models compactly store and express plain models for each variant processed by the variational solver *without* storing a plain model for each variant. Variational models do this by representing each variant by its variation context and mapping each variable in the query formula to a Boolean formula of variation contexts in disjunctive normal form. Plain models for a particular variant are then recovered through substitution on the variational model with the variant's configuration.

4.2 Formalization

In this subsection, we formalize variational SAT solving by specifying the semantics of the *accumulation*, *evaluation*, and solving the variational core, which we call *choice removal*. As described in Figure 3, the variational solver interacts with the base solver via several primitive operations. In our semantics, we simulate these primitive operations by tracking their effects on two stores. The *accumulation store* Δ is a map from IL terms to symbolic references, which tracks the values cached during accumulation. The *evaluation store* Γ is the set of symbolic references that have been sent to the base solver during evaluation.

Interface to the Base Solver. Figure 7 lists a minimal set of primitive operations that the base solver is assumed to support. These primitive operations define the interface between the base solver and the variational solver.

The primitive operations can be roughly grouped into two categories. The first four operations, consisting of the logical operations `Not`, `And`, and `Or`, plus

`Not`	:	$\Delta \times s$	$\rightarrow$	$\Delta \times s$	*Negate a symbolic value*
`And`	:	$\Delta \times s \times s$	$\rightarrow$	$\Delta \times s$	*Conjunction of symbolic values*
`Or`	:	$\Delta \times s \times s$	$\rightarrow$	$\Delta \times s$	*Disjunction of symbolic values*
`Var`	:	$\Delta \times r$	$\rightarrow$	$\Delta \times s$	*Create symbolic value based on a variable*
`Assert`	:	$\Gamma \times \Delta \times s$	$\rightarrow$	Γ	*Assert a symbolic value to the solver*
`GetModel`	:	$\Gamma \times \Delta$	$\rightarrow$	m	*Get a model for the current solver state*

Fig. 7: Assumed base solver primitive operations.

the `Var` operation, are used in the accumulation phase and are concerned with creating and maintaining symbolic references that may stand for arbitrarily complex subtrees of the original formula. These operations simulate caching information in the base solver. The final two operations, `Assert` and `GetModel`, are used in the evaluation phase and simulate pushing new assertions to the base solver and obtaining a plain satisfiability model based on the current solver state, respectively.

It's important to note that our primitive operations are pure functions and do not simulate interacting with the base solver via side effects. The effect of a primitive operation can be determined by observing its type. For example, the `Assert` operation pushes new assertions to the base solver; its type represents this by including an evaluation store as input and producing a new evaluation store (with the assertion included) as output. Note that we do not need a primitive operation to simulate popping assertions from the base solver. Instead, we simulate this by directly reusing old evaluation stores.

Many of the primitive operations operate on references to symbolic values. Such symbolic references may stand for arbitrarily complex subtrees of the original formula, built up through successive calls to the corresponding primitive operations. For example, recall the example formula $p \wedge \neg q$ from Section 4.1, which was replaced by the symbolic value s_{pq} after the following sequence of SMTLIB2 declarations.

```
(declare-const p Bool)
(declare-const q Bool)
(declare-fun s_pq () Bool (and p (not q)))
```

In our formalization, we would represent this same transformation of the formula $p \wedge \neg q$ into a symbolic reference s_{pq} using the following sequence of primitive operations:

$$\begin{aligned}
\texttt{Var}(\varnothing, p) &= (\Delta_1, s_p) \\
\texttt{Var}(\Delta_1, q) &= (\Delta_2, s_q) \\
\texttt{Not}(\Delta_2, s_q) &= (\Delta_3, s'_q) \\
\texttt{And}(\Delta_3, s_p, s'_q) &= (\Delta_4, s_{pq})
\end{aligned}$$

The accumulation store tracks the information that is associated with each symbolic reference. The store must therefore be threaded through the calls to each primitive operation so that subsequent operations have access to existing definitions and can produce a new, updated store. For example, the final store

$$\underline{\texttt{Var}}(\Delta, r) = \begin{cases} (\Delta, s) & (r, s) \in \Delta \\ \texttt{Var}(\Delta, r) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Not}}(\Delta, s) = \begin{cases} (\Delta, s') & (\neg s, s') \in \Delta \\ \texttt{Not}(\Delta, s) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{And}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \wedge s_2, s_3) \in \Delta \\ \texttt{And}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Or}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \vee s_2, s_3) \in \Delta \\ \texttt{Or}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

Fig. 8: Wrapped accumulation primitive operations.

produced by the above example contains the following mappings from IL terms to symbolic references, $\Delta_4 = \{(p, s_p), (q, s_q), (\neg s_q, s'_q), (s_p \wedge s'_q, s_{pq})\}$.

When comparing the SMTLIB2 notation to our formalization, observe that each use of declare-const corresponds to a use of the `Var` primitive, while the declare-fun line in SMTLIB2 may potentially expand into several primitive operations in our formalization. For the evaluation primitives, the `Assert` operation corresponds to an SMTLIB2 assert call, while the `GetModel` operation corresponds roughly to an SMTLIB2 check-sat call, which retrieves a (plain) model for the current set of assertions on the stack. However, the exact semantics of check-sat depends on the base solver in use. For example, given the plain formula $p = a \vee b \vee c$, the Z3 solver returns only a minimal satisfiable model, such as $\{b = \texttt{T}\}$, providing no values for the other variables in the formula. To normalize this behavior across solvers, we instead consider `GetModel` equivalent to check-sat followed by a get-value call for every variable in the query formula, yielding a complete model. For example, a complete model for p would be $\{a = \texttt{F}, b = \texttt{T}, c = \texttt{F}\}$.

Finally, in Figure 8 we define wrapped versions of the primitive operations used in accumulation to cache already computed values. These wrapper functions first check to see whether a symbolic reference for the given IL term exists already in the accumulation store, and if so, returns it without changing the store. Otherwise, it invokes the corresponding primitive operation to generate and return the new symbolic reference and updated store.

Accumulation. The accumulation phase is defined inductively using inference rules (Harper, 2016, Section 2.2) in Figure 9 as a relation of the form $(\Delta, v) \mapsto (\Delta', v')$. Accumulation replaces plain subterms with references to symbolic values whenever possible. The heart of accumulation are the first four rules in Figure 9. In the A-Ref rule, a variable reference is replaced by a symbolic reference by invoking the wrapped version of the `Var` primitive, which returns the corresponding symbolic reference or generates a new one, if needed.

The A-Not-S, A-And-S, and A-Or-S rules all replace an IL term by a symbolic reference by invoking the corresponding wrapped primitive operation.

$$\frac{\underline{\texttt{Var}}(\Delta, r) = (\Delta', s)}{(\Delta, r) \mapsto (\Delta', s)}\ \textsc{A-Ref} \qquad \frac{(\Delta, v) \mapsto (\Delta', s) \qquad \underline{\texttt{Not}}(\Delta', s) = (\Delta'', s')}{(\Delta, \neg v) \mapsto (\Delta'', s')}\ \textsc{A-Not-S}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, s_1) \qquad (\Delta_1, v_2) \mapsto (\Delta_2, s_2) \qquad \underline{\texttt{And}}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)}{(\Delta, v_1 \wedge v_2) \mapsto (\Delta_3, s_3)}\ \textsc{A-And-S}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, s_1) \qquad (\Delta_1, v_2) \mapsto (\Delta_2, s_2) \qquad \underline{\texttt{Or}}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)}{(\Delta, v_1 \vee v_2) \mapsto (\Delta_3, s_3)}\ \textsc{A-Or-S}$$

$$\frac{}{(\Delta, D\langle e_1, e_2\rangle) \mapsto (\Delta, D\langle e_1, e_2\rangle)}\ \textsc{A-Chc} \qquad \frac{(\Delta, v) \mapsto (\Delta', v')}{(\Delta, \neg v) \mapsto (\Delta', \neg v')}\ \textsc{A-Not-V}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, v_1') \qquad (\Delta_1, v_2) \mapsto (\Delta_2, v_2')}{(\Delta, v_1 \wedge v_2) \mapsto (\Delta_2, v_1' \wedge v_2')}\ \textsc{A-And-V}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, v_1') \qquad (\Delta_1, v_2) \mapsto (\Delta_2, v_2')}{(\Delta, v_1 \vee v_2) \mapsto (\Delta_2, v_1' \vee v_2')}\ \textsc{A-Or-V}$$

Fig. 9: Accumulation inference rules.

These rules all require that their subterms reduce to symbolic references. This invariant is enforced by a premise in each rule which recursively accumulates the subterm to a symbolic. For example, the premise $(\Delta, v) \mapsto (\Delta', s)$ in the A-Not-S rule enforces that the subterm being negated v, can be accumulated to a symbolic s, otherwise the `Not` cannot be invoked. Similar premises occur in A-Not-S A-And-S and A-Or-S to guarantee this invariant. Observe that a left-to-right, post-order evaluation is enforced in these rules by the propagation of the accumulation store. For example, in A-And-S, the accumulation of v_1 yields the store Δ_1, which is used as input to the accumulation of v_2; the accumulation of v_2 produces Δ_2, which is passed to the wrapped primitive $\underline{\texttt{And}}$ operation.

However, not all subterms can be completely reduced to symbolic references. In particular, variational subterms—subterms that contain one or more choices within them—cannot be accumulated to a symbolic reference. The A-Chc rule prevents accumulation under a choice. The A-Not-V, A-And-V, and A-Or-V rules are congruence rules that recursively apply accumulation to subterms. Although not explicitly stated in the premises, it is assumed that these A-*-V rules apply only if the corresponding A-*-S rule does not apply. That is, when at least one of the subterms does not reduce completely to a symbolic reference (because it contains a choice).

We have omitted rules for processing the constant values `T` and `F`. These rules correspond closely to the A-Ref rule, but use a predefined variable reference for the true and false constants.

To illustrate the semantics of accumulation, consider the plain formula $g = a \vee (a \wedge b)$ with an initial accumulation store $\Delta = \varnothing$. The A-Or-S rule matches the root $\vee$ connective with $v_1 = a$ and $v_2 = a \wedge b$. Since v_1 is a reference, the

$$\frac{(\varDelta, v) \mapsto (\varDelta', v') \qquad (\varGamma, \varDelta', v') \rightarrowtail (\varGamma', \varDelta'', v'')}{(\varGamma, \varDelta, v) \rightarrowtail (\varGamma', \varDelta'', v'')} \textsc{E-Acc} \qquad \frac{\texttt{Assert}(\varGamma, \varDelta, s) = \varGamma'}{(\varGamma, \varDelta, s) \rightarrowtail (\varGamma', \varDelta, \bullet)} \textsc{E-Sym}$$

$$\frac{}{(\varGamma, \varDelta, D\langle e_1, e_2\rangle) \rightarrowtail (\varGamma, \varDelta, D\langle e_1, e_2\rangle)} \textsc{E-Chc} \qquad \frac{}{(\varGamma, \varDelta, v_1 \vee v_2) \rightarrowtail (\varGamma, \varDelta, v_1 \vee v_2)} \textsc{E-Or}$$

$$\frac{(\varGamma, \varDelta, v_1) \rightarrowtail (\varGamma_1, \varDelta_1, \bullet) \qquad (\varGamma_1, \varDelta_1, v_2) \rightarrowtail (\varGamma_2, \varDelta_2, v_2')}{(\varGamma, \varDelta, v_1 \wedge v_2) \rightarrowtail (\varGamma_2, \varDelta_2, v_2')} \textsc{E-And-L}$$

$$\frac{(\varGamma, \varDelta, v_1) \rightarrowtail (\varGamma_1, \varDelta_1, v_1') \qquad (\varGamma_1, \varDelta_1, v_2) \rightarrowtail (\varGamma_2, \varDelta_2, \bullet)}{(\varGamma, \varDelta, v_1 \wedge v_2) \rightarrowtail (\varGamma_2, \varDelta_2, v_1')} \textsc{E-And-R}$$

$$\frac{(\varGamma, \varDelta, v_1) \rightarrowtail (\varGamma_1, \varDelta_1, v_1') \qquad (\varGamma_1, \varDelta_1, v_2) \rightarrowtail (\varGamma_2, \varDelta_2, v_2')}{(\varGamma, \varDelta, v_1 \wedge v_2) \rightarrowtail (\varGamma_2, \varDelta_2, v_1' \wedge v_2')} \textsc{E-And}$$

Fig. 10: Evaluation inference rules.

A-Ref rule applies, generating a new symbolic reference s_a and store $\varDelta_1 = \{(a, s_a)\}$ via $\underline{\texttt{Var}}(\varDelta, a) = (\varDelta_1', s_a)$. Processing v_2 requires an application of the A-And-S rule with $v_1' = a$ and $v_2' = b$, both of which require another application of the A-Ref rule. For v_1', the variable a is already present in the store $\varDelta_1$ since s_a was previously generated, so s_a is returned without modifying the store (because we use the wrapper primitive $\underline{\texttt{Var}}$). For v_2', a new symbolic reference s_b is generated and added to the store yielding $\varDelta_2 = \{(a, s_a), (b, s_b)\}$. Since both the left and right sides of v_2 reduce to a symbolic reference, the `And` primitive is invoked, yielding a new symbolic reference s_{ab} and the store $\varDelta_3 = \{(a, s_a), (b, s_b), (a \wedge b, s_{ab})\}$. Finally, since both the left and right sides of g reduce to symbolic references, the `Or` primitive is invoked yielding the final symbolic reference s_g, and the final accumulation store $\varDelta_4 = \{(a, s_a), (b, s_b), (s_a \wedge s_b, s_{ab})(s_a \vee s_{ab}, s_g)\}$.

When a formula contains choices, all of the plain subterms surrounding the choices are accumulated to symbolic references, but choices remain in place and their alternatives are not accumulated. For example, consider the variational formula $g' = (a \vee (a \wedge b)) \vee D\langle a, a \wedge b\rangle \wedge (a \vee (a \wedge b))$ which contains two instances of g as subterms. The formula g' accumulates to the variational core $s_g \vee D\langle a, a \wedge b\rangle \wedge s_g$ with the same final store $\varDelta_4$ produced when accumulating g alone. Note that each instance of g in g' was reduced to the same symbolic reference s_g and the alternatives of the choice were not reduced.

Evaluation. The evaluation phase is defined in Figure 10 as a relation of the form $(\varGamma, \varDelta, v) \rightarrowtail (\varGamma', \varDelta', v')$, where an evaluation store $\varGamma$ represents the base solver's state. The E-Acc and E-Sym rules are the heart of evaluation: E-Acc enables accumulating subterms during evaluation, while E-Sym sends a fully accumulated subterm to the base solver. The E-Acc rule passes the input term v to accumulation (note the different arrows in the premises; the first premise links to the accumulation relation in Figure 9), then re-evaluates the accumulated term v'. Thus, the E-Acc rule enables alternating between accumulation and evaluation until the term is fully evaluated.

Once a term has been fully accumulated to a symbolic value, it can be sent to the base solver by E-Sym. When a subterm is sent to the base solver (using `Assert`), it is replaced by the unit value $\bullet$ and the evaluation store Γ is updated accordingly. Conceptually, the evaluation store Γ represents the internal state of the underlying solver (e.g. Z3's internal state). We model Γ as the set of assertions that have been sent to the base solver. For example, given the accumulation store $\Delta = \{(a, s_a), (b, s_b), (s_a \wedge s_b, s_{ab})\}$, the assertion $\texttt{Assert}(\{\}, \Delta, s_a)$ yields $\{s_a\}$ as s_a is now on the base solver's assertion stack. Subsequent assertions add more elements to this set, for example, $\texttt{Assert}(\{s_a\}, \Delta, s_{ab}) = \{s_a, s_{ab}\}$.

Evaluation cannot occur under choices or un-accumulated disjunctions (i.e., disjunctions that contain choices). This is enforced by the E-Chc and E-Or rules which preserve choices and disjunctions for choice removal or accumulation. However, evaluation can occur under partially accumulated conjunctions. The reason that partially accumulated conjunctions may be evaluated but partially accumulated disjunctions may not, is that assertions on the base solver's assertion stack are implicitly conjuncted together, as described in Section 4.1.

The three E-And* rules propagate accumulation over conjunctions. In all three rules, the subterms are evaluated left-to-right, propagating the resulting stores accordingly. The E-And-L rule states that if the left side of a conjunction can be fully evaluated to $\bullet$, then the expression can be evaluated to the result of the right side. Likewise, E-And-R states that if the right side fully evaluates, the result of evaluating the expression is the result of the left side. If neither side fully evaluates to $\bullet$ (because both contain choices or disjunctions), E-And applies, which evaluates subterms but leaves the conjunction in place to be handled during choice removal.

As an example, consider evaluating the formula $g = (a \vee b) \wedge D\langle a, c\rangle$ with initially empty stores. We start by applying accumulation using the E-Acc rule, yielding the intermediate term $g' = s_{ab} \wedge D\langle a, c\rangle$ with the accumulation store $\Delta = \{(a, s_a), (b, s_b), (s_a \vee s_b, s_{ab})\}$. We then apply E-And-L to g', which sends the left subterm s_{ab} to the base solver via the E-Sym rule, and the right side will be unevaluated via the E-Chc rule resulting in the expression $\bullet \wedge D\langle a, c\rangle$. Ultimately, evaluation yields the expression $D\langle a, c\rangle$ via the E-And-L rule, with accumulation store Δ and evaluation store $\Gamma = \{s_{ab}\}$.

Choice removal. The top-level relation and main driver of variational solving is the choice removal phase, which is defined in Figure 11 as a relation of the form $(C, \Gamma, \Delta, M, z, v) \Downarrow M'$. The main role of choice removal is to relate an IL term v to a variational model M'. However, doing this requires several pieces of context including a configuration C, an evaluation store Γ, an accumulation store Δ, an initial variational model M, and an evaluation context z. The two stores have been explained earlier in this subsection, and are only maintained by choice removal so they can be passed to the corresponding evaluation and accumulation phases. Variational models are explained at the

$$\frac{(\mathit{\Gamma}, \mathit{\Delta}, v) \rightarrowtail (\mathit{\Gamma}', \mathit{\Delta}', \bullet) \qquad \mathit{Combine}(M, \texttt{GetModel}(\mathit{\Delta}, \mathit{\Gamma})) = M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, \top, v) \Downarrow M'}\ \textsc{C-Eval}$$

$$\frac{(D, \mathsf{true}) \in C \qquad (C, \mathit{\Gamma}, \mathit{\Delta}, M, z, e_1) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, z, D\langle e_1, e_2\rangle) \Downarrow M'}\ \textsc{C-Chc-T}$$

$$\frac{(D, \mathsf{false}) \in C \qquad (C, \mathit{\Gamma}, \mathit{\Delta}, M, z, e_2) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, z, D\langle e_1, e_2\rangle) \Downarrow M'}\ \textsc{C-Chc-F}$$

$$\frac{\begin{array}{c} D \notin \mathit{dom}(C) \\ (C \cup (D, \mathsf{true}), \mathit{\Gamma}, \mathit{\Delta}, M, z, e_1) \Downarrow M_1 \qquad (C \cup (D, \mathsf{false}), \mathit{\Gamma}, \mathit{\Delta}, M_1, z, e_2) \Downarrow M_2 \end{array}}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, z, D\langle e_1, e_2\rangle) \Downarrow M_2}\ \textsc{C-Chc}$$

$$\frac{(C, \mathit{\Gamma}, \mathit{\Delta}, M, \neg\cdot :: z, v) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, z, \neg v) \Downarrow M'}\ \textsc{C-Not}$$

$$\frac{(\mathit{\Delta}, \neg s) \mapsto (\mathit{\Delta}', s') \qquad (C, \mathit{\Gamma}, \mathit{\Delta}, M, z, s') \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, \neg\cdot :: z, s) \Downarrow M'}\ \textsc{C-Not-In}$$

$$\frac{(C, \mathit{\Gamma}, \mathit{\Delta}, M, \cdot \wedge v_2 :: z, v_1) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, z, v_1 \wedge v_2) \Downarrow M'}\ \textsc{C-And} \qquad \frac{(C, \mathit{\Gamma}, \mathit{\Delta}, M, s \wedge \cdot :: z, v) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, \cdot \wedge v :: z, s) \Downarrow M'}\ \textsc{C-And-InL}$$

$$\frac{(\mathit{\Delta}, s_1 \wedge s_2) \mapsto (\mathit{\Delta}', s_3) \qquad (C, \mathit{\Gamma}, \mathit{\Delta}, M, z, s_3) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, s_1 \wedge \cdot :: z, s_2) \Downarrow M'}\ \textsc{C-And-InR}$$

$$\frac{(C, \mathit{\Gamma}, \mathit{\Delta}, M, \cdot \vee v_2 :: z, v_1) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, z, v_1 \vee v_2) \Downarrow M'}\ \textsc{C-Or} \qquad \frac{(C, \mathit{\Gamma}, \mathit{\Delta}, M, s \vee \cdot :: z, v) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, \cdot \vee v :: z, s) \Downarrow M'}\ \textsc{C-Or-InL}$$

$$\frac{(\mathit{\Delta}, s_1 \vee s_2) \mapsto (\mathit{\Delta}', s_3) \qquad (C, \mathit{\Gamma}, \mathit{\Delta}, M, z, s_3) \Downarrow M'}{(C, \mathit{\Gamma}, \mathit{\Delta}, M, s_1 \vee \cdot :: z, s_2) \Downarrow M'}\ \textsc{C-Or-InR}$$

Fig. 11: Choice removal inference rules

end of Section 4.1. We explain configurations and evaluation contexts when describing the relevant rules below.

The C-Eval rule is our base case, and states that if v evaluates to $\bullet$ (because all choices have been removed and we are left with a plain term), then we can get the current model from the base solver using the `GetModel` primitive and update our variational model. We use the operation *Combine* to perform the variational model update operation described in Section 4.1.[7] The *Combine* operation merges the plain model obtained from `GetModel` with the current variational model M yielding an updated variational model M'. The rest of the choice removal rules are structured so that C-Eval will be invoked once for every variant of the variational core. The final output will be a variational model that encodes the solutions to every variant of the original formula.

[7] Note that *Combine* is not a primitive operation of the base solver, but rather a function in the variational solver that updates a variational model.

The three C-Chc* rules concern choices and are the heart of choice removal. These rules make use of a *configuration* C, which maps dimensions to Boolean values or $\bot$ and is encoded as a set of pairs. The configuration tracks which dimensions have been selected and ensures that all choices in the same dimension are synchronized (see Section 3). Whenever a choice $D\langle e_1, e_2\rangle$ is encountered during choice removal, we check the configuration C to determine how to resolve the choice. In C-Chc-T, if $(D, \mathsf{true}) \in C$, then the first alternative of the dimension has already been selected, so choice removal proceeds on e_1. Similarly in C-Chc-F, if $(D, \mathsf{false}) \in C$ then the right alternative has been selected, so choice removal proceeds on e_2. In C-Chc if $D \notin \mathit{dom}(C)$ and thus $C(D) = \bot$, then no value for the dimension D has yet been selected, so we recursively apply choice removal to both e_1 and e_2, updating C accordingly in each case. Observe that we use the same accumulation store Δ, evaluation store Γ, and evaluation context z for each alternative. This represents a backtracking point in the solver, where we first solve e_1, then reset the state of the solver to the point where we encountered the choice and solve e_2. Only the variational model, which is threaded through the solution of both e_1 and e_2, is maintained to accumulate the results of solving each alternative.

The remaining eight rules apply choice removal to the usual logical operations. These rules make heavy use of an *evaluation context* z that keeps track of where we are in a partially evaluated IL term during choice removal. Evaluation contexts are defined as a zipper data structure (Huet, 1997) over IL terms, given by the following grammar:

$$z \ ::= \ \top \mid \neg\cdot :: z \mid \cdot \wedge v :: z \mid s \wedge \cdot :: z \mid \cdot \vee v :: z \mid s \vee \cdot :: z$$

An evaluation context z is a data structure that enables focusing on a subterm within a partially evaluated IL term. A helpful metaphor is to think of z as a "breadcrumb trail", where new breadcrumbs are added to the left and separated from the rest of the trail by the :: symbol, and where each breadcrumb tracks the work that we've done and the work we have yet to do. The empty context $\top$ indicates the root of the term. The other cases in the grammar prepend a crumb to the trail. The crumb $\neg\cdot$ focuses on the subterm within a negation, $\cdot \wedge v$ focuses on the left subterm within a conjunction whose right subterm is an unevaluated term v (work left to do), and $s \wedge \cdot$ focuses on the right subterm of a conjunction whose left subterm has already been reduced to s. The cases for disjunction are similar to conjunction.

As an example, consider the IL term $\neg(a \vee b) \wedge c$. When evaluation is focused on a, the evaluation context will be $\cdot \vee b :: \neg\cdot :: \cdot \wedge c :: \top$, which states that a exists as the left child of a disjunction whose right child is b, which is inside a negation, which is the left child of a conjunction whose right child is c. The b and c terms captured in the context are subterms of the original term that must still be evaluated. During choice removal, IL terms are evaluated according to a left-to-right, post-order traversal; as IL subterms are evaluated they are replaced by symbolic references via accumulation. When evaluation is focused on b, the context will be $s_a \vee \cdot :: \neg\cdot :: \cdot \wedge c :: \top$, where s_a is the symbolic reference produced by accumulating the variable a, and where the rest of the evaluation context is unchanged since we have not left evaluation of the innermost disjunction yet. When evaluation is eventually focused on c, the evaluation context will be simply $s_{ab} \wedge \cdot :: \top$ since the entire subtree $\neg(a \vee b)$ on the left side of the conjunction will have been accumulated to the symbolic reference s_{ab}.

The C-Not, C-And, and C-Or rules define what to do when encountering a logical operation for the first time. In C-Not, we focus on the subterm of the negation, while in C-And and C-Or, we focus on the left child while saving the right child in the context.

The C-And-InL and C-Or-InL rules define what to do when *finished* processing the left child of the corresponding operation.[8] A fully processed child has been accumulated to a symbolic reference s. At this point, we move the s into the evaluation context and shift focus to the previously saved right child of the logical operation.

Finally, the C-Not-In, C-And-InR, and C-Or-InR rules define what to do when evaluation finished processing all children of a logical operation. At this point, all children will have been reduced to symbolic references (s, s_1, and s_2 respectively) so we can accumulate the entire subterm and apply choice removal to the result. For example, in C-And-InR, we have just finished processing the right child to s_2 and we previously reduced the left child to s_1, so in the first premise we now accumulate $s_1 \vee s_2$ to s_3. In the second premise, we proceed with choice removal by recursively applying the relation to the accumulated term s_3 in the parent context z (i.e., the context in which the original term was found).

Evaluation contexts support solving variational formulas recursively by adding to the context as we move down the term and removing from the context as we move back up. The extra effort over a more direct recursive strategy is necessary to support the backtracking pattern implemented by the C-Chc rule. Whenever we encounter a choice in a new dimension, we can simply split the state of the solver to explore each alternative. Without evaluation contexts, this would be extremely difficult since choices may be deeply nested within a variational formula. We would have to somehow remember all of the locations in the term that we must backtrack to later and the state of the solver at each of those locations. Instead, we make this context explicit as z and use two copies of it, one in each premise of C-Chc.

Summary. Together, the definitions of accumulation ($\mapsto$), evaluation ($\rightarrowtail$), and choice removal ($\Downarrow$) formalize variational SAT solving. Variational solving is driven by choice removal, which invokes evaluation, which in turn invokes accumulation. The interaction of these phases was informally described in

[8] The -InL and -InR rule name suffixes are short for "in the left subterm" and "in the right subterm", respectively. So the rule C-And-InL applies when we are inside (and finished processing, since the focused expression is a symbolic reference s) the left subterm of a conjunction, while C-And-InR applies when we are inside the right subterm of a conjunction.

Section 4.1, but are formally defined by the rules in Figure 9, Figure 10 and Figure 11 that link these phases together via their premises.

Accumulation replaces plain subterms by symbolic values, maximizing sharing and reuse during variational solving. Evaluation guides the accumulation process and coordinates the interaction with the base solver by sending accumulated subterms to the assertion stack. Finally, choice removal implements a backtracking traversal of the variational formula, configuring choices, evaluating subterms, and updating the variational model as each plain variant is encountered.

5 Quantitative Evaluation

Section 4 formalizes an approach to variational solving that shares subterms in various ways to reduce the work of solving a large number of variant formulas. However, theoretical solutions may not translate to observable effects in real-world workloads, especially in the domain SAT and SMT solvers, which are notoriously hard to evaluate using synthetic data (Gent and Walsh, 1994). In this section, we provide a quantitative evaluation using a prototype variational SAT solver to validate that our approach translates to reduced execution time when solving sets of related SAT problems. Section 5.1 describes our experiment methodology and Section 5.2 presents and discusses our results.

5.1 Experimental Methodology

There are several questions of interest to investigate. First, we must know whether our two-step approach of generating and then solving a variational core does reduce the solver execution time when solving many variants, and therefore when workloads are highly variational. Second, we seek to answer what effect the base solver has on the execution time of a variational SAT solver. It is well known that the performance of incremental and non-incremental SAT solvers varies even when run on the same input (Balyo et al., 2020) and thus we expect the base solver to have an effect on execution time on variational solving. Third, we must understand the effect sharing has on the execution time of a variational SAT solver. Key to our approach is caching equal subterms through indirection with symbolic values. Thus, confirming that sharing positively impacts execution time would validate our approach. Fourth, we investigate if and when variational solving pays off performance-wise over a manual operation of incremental solvers. While a key qualitative benefit of variational SAT solving is freeing users from having to hand-write instructions for incremental solvers, it may also pay off in performance. In particular, we wish to determine a threshold of variants above which variational solving is faster than incremental solving (neglecting the overhead for instructing the incremental solver). Lastly, we investigate the execution time slowdown induced by our approach to variational solving outside of its intended use case, i.e., when solving plain VPL formulas. Our approach incurs an extra pre-processing cost under the assumption this cost pays off over many variants. Observing and estimating this cost elucidates future avenues of work to mitigate this cost.

We investigate each research question with our prototype solver VSAT, which we present in detail in Sec. 5.1.1. We summarize our research questions as follows:

RQ1 How does execution time of VSAT increase as the number variants represented in a VPL formula increases?
RQ2 What is the impact of the base solver on execution time of VSAT?
RQ3 What is the impact of sharing of plain terms on execution time of VSAT?
RQ4 What is the cost of solving a plain formula on VSAT?
RQ5 What is the threshold of variants to solve, such that, VSAT's execution time is reduced compared to directly operating incremental solvers?

The rest of this section presents our experimental methodology. We describe the solving strategies used in our comparisons in Sec. 5.1.2, describe the data used in the evaluation in Sec. 5.1.3, and describe the statistical methods used to detect differences in the response data in Sec. 5.1.4. Finally, in Sec. 5.1.5, we motivate and establish the experimental setup for each research question in detail.

5.1.1 Prototype Implementation

We construct two systems to answer our research questions; a prototype variational solver and a benchmarking system for off-the-shelf SAT solvers. Our prototype variational solver, called VSAT, is written in the Haskell programming language (Hudak et al., 1992). VSAT utilizes a widely used Haskell library called sbv (Erkok, 2011) to interface and instruct its base solver. The sbv library provides a generic interface to SMTLIB2 conforming solvers, and therefore possible base solvers; specifically Z3 (de Moura and Bjørner, 2008), CVC4 (Barrett et al., 2011), Boolector (Brummayer and Biere, 2009), and Yices (Dutertre, 2014).

The benchmarking system is a thin wrapper over the sbv library that selects and manages the solver instance being benchmarked. Sbv exposes configuration settings which run each base solver either incrementally, or non-incrementally. We exploit these settings in the benchmarking system to craft *non-variational* benchmarks to compare VSAT to. Therefore, this design (i.e., using sbv for the base solver communication in VSAT and for benchmarking non-variational solvers) allows us to maintain the same interface to each off-the-shelf solver, while comparing VSAT to each solver run either incrementally or non-incrementally.

5.1.2 Solving Algorithms

We define four different solving algorithms that input the same VPL formula but solve it differently, thereby constituting the subject of comparison in the experiment. We use the notation

$$\langle \textit{formula} \rangle \to \langle \textit{solver} \rangle$$

to describe, for each algorithm, whether the query formulas and solvers are plain (p) or variational (v), respectively. The algorithms are: the brute force case, $p \to p$, the baseline case, $v \to p$, the variational case, $v \to v$, and the slowdown case, $p \to v$.

The $p \to p$ case solves a set of plain formulas (p) on a plain *non-incremental* solver (p). We construct the $p \to p$ algorithm by configuring the query VPL formula to its variants *before* benchmarking begins. These formulas are then sent to the non-incremental solver one-by-one. The solver is shut down and re-initialized between runs, i.e., after solving each variant, to prevent the solver from maintaining any learned information. The $p \to p$ algorithm gives insight into the execution time of the non-incremental solvers if no solver information and no terms are shared between SAT problems. This algorithm represents the worst-case scenario.

The baseline case $v \to p$ runs a variational formula (v), variant by variant on a plain *incremental* solver (p). The algorithm configures the query formula to retrieve variants *during* benchmarking. Each formula is sent to the base solver with the solver maintaining information between queries. This gives insight into the slowdown incurred by configuring a variational formula, and the benefits of the internal caching in the base solver. This method corresponds to approaches observed in the aforementioned variational systems; where sharing is exploited through use of an incremental solver but not increased by our more granular approach. This algorithm represents the experimental control group.

The variational case $v \to v$ runs a VPL formula (v) on VSAT (v); it is the subject of interest in most of the comparisons in the experiment.

Lastly, the slowdown case $p \to v$ solves a set of plain formulas (p) on the variational solver (v). We perform the same pre-processing as for the $p \to p$ case (i.e., configuring the query formula to its variants) *before* benchmarking begins but send each plain formula to VSAT instead of a plain solver. Since the input to VSAT in this algorithm is plain, VSAT will accumulate the entire formula to a single symbolic and then assert it into the base solver. This algorithm provides insight into the cost incurred by the reduction engine because only the reduction engine will process the plain input to a single symbolic. The rest of the runtime will be internal to the base solver.

We construct a variational model for all algorithms because (1) we are then able to provide descriptive statistics about the variational models from real-world data, and (2) the storage of plain models is an orthogonal concern to performance.

5.1.3 Data Description and Encoding

We compare VSAT to off-the-shelf incremental and non-incremental SAT solvers using real-world data from a previous study by Nieke et al. (2018). Nieke et al. provide two real-world datasets: *automotive02* and *financialServices1*, which encode the evolution histories of two feature models as propositional formulas. [9] We refer to these as the *auto* and *fin* dataset for the remainder of the paper. Nieke et al.'s formulas collapse sets of C_2 formulas to a single formula using material implication on an SMT variable that represents a discrete moment in time. A two-pass process was used to translate Nieke et al.'s formulas into a single VPL formula each—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, and placing a `T` in the false alternative. The two-pass process conserves Nieke et al.'s ordering of plain terms and encoding. The two datasets differ in their amount of variation. 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.[10]

5.1.4 Measuring Performance

Unless specified, all results are a bootstrapped statistical average representing three raw measurements.[11] To test for statistical difference between algorithms we perform a Kruskall-Wallis test (National Institute of Standards and Technology, 2020) followed by a pairwise Wilcox test (Wilcoxon, 1945) with Holm-Bonferroni p-value correction (Holm, 1979) in the R programming language (R Core Team, 2020) v4.0.3 and assume a 5% significance level.

5.1.5 Research Questions in Detail

Having defined our subject systems, the algorithms used for comparison, and the evaluation dataset, we now turn to an in-depth discussion of each research question.

RQ1: Execution time as variants increase. The motivation for RQ1 is to verify that our approach is viable on realistic workloads; when the number of variants is so high that hand writing the incremental solving procedure is difficult. Theoretically, we would expect that as the number of variants to solve increases, the more effective our method of variational solving should become because more reuse of shared plain terms would occur. Thus, we expect to observe the total execution time of the variational solver to reduce compared to incremental and non-incremental solvers when solving sets of SAT problems expressed as a VPL formula.

[9] `https://gitlab.com/evolutionexplanation/evolutionexplanation`

[10] See the `src/CaseStudy` directory in `https://doi.org/10.5281/zenodo.5543884` for the implementation.

[11] Using v0.2.5 of the gauge (O'Sullivan, 2009) library and v8.6 of the sbv (Erkok, 2011) library with solver seeds set to `1729`. All data was collected on a desktop running NixOS 20.09, with an AMD Ryzen 7 2700X CPU @ 4.0GHz, 32GB RAM. We used stack lts-15.7 (GHC 8.8.3), tested with RTS options "-qg" which enables parallel garbage collection and disable frequency scaling of the CPU.

The experimental setup for RQ1 is a statistical comparison of execution time (wall time) between $v \to v$, and each other algorithm as the number of variants to solve increases. The key comparison is between the variational case $v \to v$, and the baseline case $v \to p$. Should $v \to v$ show reduced execution time compared to $v \to p$ as variants increase, then we can conclude our method of variational solving is performant compared to the baseline case. The comparison is a direct test of our methods, i.e., generating variational cores and then solving them. $v \to p$ incrementally solves by configuring the query formula but does not use accumulation and evaluation, instead it configures and then directly solves the variant. Thus, differences in execution time between $v \to v$ and $v \to p$ are representative of time spent on accumulation, evaluation, and solving a variational core.

To estimate the impact as variant count increases we fit the linear model `Execution Time` $= c$ `*` `Variant Count` to the bootstrapped data where c is the scaling constant factor. We report the adjusted coefficient of determination for the model and the model's estimate of the scaling factor for each algorithm. There are two comparisons and thus two p-values. The first p-value verifies that the model's results are meaningful by checking if the model fits the data more accurately than noise. The second p-value results from testing if $c = 0$. Should this p-value be under 0.05 then we may conclude that the scaling factor is not 0 (the null hypothesis) and thus the estimate returned by the linear model is meaningful (the alternate hypothesis). All results for RQ1 were found using Z3 as the base solver.

RQ2: Impact of the base solver on execution time. SAT solvers are complex systems which utilize heuristics to optimize performance. Many solvers make trade-offs to optimize for particular problem domains. For example, the solver Boolector is optimized to solve problems common in model checking, such as SMT problems which contain quantifier free fixed-size bit-vectors (Barrett et al., 2016), and has consistently beat other SMT solvers in this area (Brummayer and Biere, 2021). Therefore, we expect that certain solvers are more performant for variational workloads than other solvers. Consequently, we expect that the execution time of VSAT will be affected by its base solver. Should this effect exist then it would allow future implementations to choose a base solver with good performance for their particular domain. Similarly, should the effect be large, then it may indicate that a useful feature for future variational solvers is the ability for a user to pick the base solver for their domain, or for a subset of variants.

The experimental setup for RQ2 is the comparison of execution time of $v \to v$ to itself with different base solvers (i.e., the solvers supported by the sbv library). Additionally, we compare the brute force case ($p \to p$), and the baseline case ($v \to p$) with the supported solvers to observe the impact on execution time for non-variational solvers.

RQ3: Impact of sharing on execution time. Previous research on variational data structures (Meng et al., 2017; Smeltzer and Erwig, 2017) and a variational bytecode interpreter (Meinicke, 2014) observed that the amount of sharing, and the distribution of variation, had an impact on execution time of these systems. The *sharing hypothesis* is that execution time decreases when the degree of sharing increases, because the variational system can reuse more information when processing variants. Similarly, the *distribution hypothesis* is, for a given domain, if contiguous segments of plain domain elements are separated by few variational elements, then execution time decreases because this grouping naturally allows more reuse across variants. For variational SAT solving, the distribution hypothesis equates to slowdowns if the VPL formula interleaves choices and *single* plain terms. For example, $A\langle \ldots, \ldots \rangle \wedge\ p_0\ \wedge B\langle \ldots, \ldots \rangle \wedge\ p_1\ \ldots$ where p_i are plain terms. A well formulated formula would have groupings of plain terms, which are easier to accumulate, and groupings of variational terms, for example, $A\langle \ldots, \ldots \rangle \wedge p_0 \wedge p_1 \wedge \ldots \wedge p_i \wedge B\langle \ldots, \ldots \rangle \wedge p_{i+1} \wedge p_{i+2} \cdots \wedge p_n$.

Our motivation for RQ3 is to test the sharing hypothesis in the domain of variational SAT solving. Theoretically, a high degree of sharing should produce a smaller variational core because segments of plain terms would reduce to a single symbolic term, and thus lower the runtime costs of solving many variants. Furthermore, if this effect is observed with VSAT, then knowing its magnitude could lead to useful optimizations and motivate further experimentation to test the distribution hypothesis. For example, one could calculate the sharing ratio of the query formula during parsing, then if the ratio is above a certain threshold the solver could re-order the query formula to improve execution time.

The experimental setup for RQ3 requires calculating the sharing ratio of query formulas and limiting the variants to solve to only variants which are not artificial mixtures of Nieke et al.'s feature model versions. There are two kinds of variants: *version variants*, which are variants that represent a specific version of a feature model, and variants which are combinations of version variants. We restrict the analysis to version variants because version variants are real-world SAT problems, and are therefore a more accurate representation of real-world workloads and sharing ratios.

We do not consider non-version variants for two reasons: First, because non-version variants are combinations of version variants they are artificial data and therefore are likely to be quickly found unsatisfiable—a well known phenomena in the SAT/SMT community (Gent and Walsh, 1994). Thus we prevent introducing bias to our results and analysis through their omission. Second, to our knowledge, the distribution of sharing ratios in real-world variational formulas is an open research question. If we included non-version variants then we would need to ensure that these variants form a valid sample, i.e., they have a distribution of sharing ratios which is representative of the population of real-world sharing ratios. Should we include version variants, then our analysis of RQ3 gains a threat to validity that our results are based on artificial VPL formulas which have a significantly different distribution of sharing ratios than VPL formulas from real-world problems.

However, there is still a major problem with restricting the analysis to version variants, namely that sharing is only defined for VPL formulas which contain choices. We cannot simply assess execution time of version variants because these variants are necessarily plain and therefore have a sharing ratio which is undefined. Instead, the sharing analysis is performed on VPL formulas which represent consecutive version variants.

We construct these formulas by transforming the original *auto* and *fin* VPL formulas (which represent every feature model version and all combination of versions) to generate a set of VPL formulas. Each VPL formula in the set only represents a consecutive pairing of feature model versions, and thus a pairing of sequential version variants. We perform a pass over the original query formula and replace choices representing non-consecutive versions variants with their false alternatives (recall that these contain the value `T`).

For example, the *auto* data set has four unique dimensions corresponding to analyses on four feature model versions. To retrieve the VPL formula that represents only the version variants V_1 and V_2, we replace all instances of V_3 and V_4 with their false alternative. This results in a VPL formula that still has choices with dimensions V_1 and V_2 but no choices with dimensions V_3 and V_4. With this new VPL formula we can then instruct each solver to solve only the true alternatives of each choice with a vc, ensuring that the solver solves the version variants of V_1 and then V_2 and nothing else. This process is repeated to produce the set of consecutive version VPL formulas for both data sets. For example, another VPL formula in the *auto* dataset would be produced by replacing V_1 and V_4 with their false alternative, yielding the VPL formula which only represents V_2 and V_3.

Next we iterate over each VPL formula in each set to calculate each formula's sharing ratio, and construct variational contexts to restrict the solvers to consecutive version variants. For example, $vc_{auto\_V12} = exactly_1(\{V_1, V_2\})$[12] restricts the solvers to the first and second versions of the *auto* dataset. By this process the *auto* dataset yields only three data points (and three consecutive version VPL formulas), the changes from versions V_1 to V_2, V_2 to V_3, and V_3 to V_4. The *fin* dataset yields nine data points.

We post process the data by normalizing execution time for a given algorithm to the baseline ($v \to p$) to calculate speedup. We statistically assess whether speedup correlates to the ratio of plain terms in query formula by fitting a linear model, and repeating the aforementioned statistical tests. All results presented for RQ3 are calculated using Z3.

RQ4: Slowdown of VSAT on plain query formulas. Thus far the research questions have focused on the impact of a sub-system (the base solver) or a property of the query formula (the sharing ratio) on the execution time. RQ4 and RQ5 differ in that they are concerned with the efficiency of VSAT. RQ4 ask how much overhead exists, and RQ5 asks when does this overhead begin to pay off. Our two-step approach should save execution time in the variational case (i.e.,

[12] See the definition of $exactly_1$ in Bittner et al. (2019).

when solving more than one variant) since information is reused over many variants. However, this same approach should cost time when solving a single variant because nothing can be reused. RQ4's purpose is to observe if this cost is detectable and determine its magnitude.

The experimental setup for RQ4 differs from previous research questions. Each input to the variational solver, and thus to $v \to v$, is a variant rather than a VPL formula. Before benchmarking begins, we configure the query formulas to produce the set of version variants for each data set. We again restrict the analysis to version variants to avoid the effect of artificial data on our results. Then, each algorithm receives the plain formulas as input rather than the VPL formula. Normally, this results in a single bootstrapped average data point per algorithm and per version variant, which is insufficient for statistical comparisons. Therefore, for RQ4, we retrieve the raw measurements from the bootstrapped average and assess statistical differences identically to RQ3 but do not fit any models to the data. All results, including variational models and statistical analysis scripts, are available online.[13]

RQ5: Observing inflection point of reduced execution time. RQ5 asks *when* does the hypothetical overhead detected in RQ4 begin to pay off. Our method of variational SAT solving requires extra time and memory to maintain and manipulate the symbolic cache, configuration, and reduce the formula to a variational core. This extra processing should initially produce a slowdown in execution time until the cost can be paid off through reuse and solving many variants. Hence, we hypothesize that there exists an inflection point in execution time for VSAT, where the number of variants to solve is so low that this cost no longer dominates and the variational solver's execution time *becomes* lower than the non-variational solvers. For example, it may be faster to simply process two variants directly with the base solver and synthesize a variational model rather than generating a variational core and solving the core. This threshold is meaningful for future implementations of variational SAT solvers, because it provides a metric to compare variational solver implementations and provides a direct path to optimizations. For instance, a future implementation could choose to use a non-variational solver if the number of variants to solve is under the threshold.

The experimental setup for RQ5 utilizes the same data and models from RQ1. Using these models, we predict the variant count at which the execution time for $v \to v$ is larger than each other algorithm. We fit a linear model in order to predict execution time by variant count for variant counts that were not directly measured in this experiment.

5.2 Results and Discussion

In this subsection, we present and discuss our results. We begin with descriptive, non-performance results, and then discuss results for each research

[13] `https://doi.org/10.5281/zenodo.5546009`

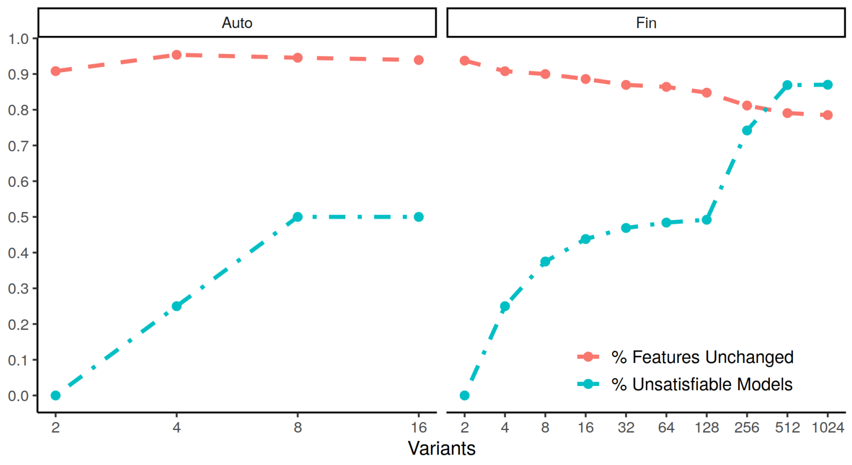


Fig. 12: Most plain models found to be unsatisfiable. Only a small portion of features ever changed to `T`.

question. We conclude the section with a discussion of when variational solving is an appropriate strategy compared to incremental solving, and threats to validity.

Non-Performance Results. This section provides descriptive statistics of the VPL encoded real-world data and concludes by reporting a substantive difference in data collection compared to the conference version. The VPL encoded data is our first glimpse at naturally occurring variation, thus we characterize the data to better understand the space of naturally occurring variational SAT problems. In particular, we calculate the sharing ratio of each formula, and calculate the ratio of constant features and satisfiable models the variational model represents. We require the sharing ratios in order to answer RQ3, and are interested in characterizing the variational models in order to inform future variational model designs. However, we do not include the variational model data in the answer for each research question because the performance of variational models is an orthogonal concern to variational SAT solving.

The datasets yielded dissimilar query formulas: the *auto* query formula consisted of 4,212 choice terms (not including terms in a choice's alternatives), and 26,808 plain terms. In contrast, *fin* had 3,809 choice terms, and 1,441 plain terms. Thus, *fin* had larger changes between product line versions. Figure 12 shows the ratio of unsatisfiable models to total plain models, and the ratio of constant features for each product line version (represented by variant count). Note that variants increase exponentially (and thus so does the x-axis) because each unique dimension in the query formula doubles the number of variants.

For both datasets, the number of satisfiable models decreased as new versions were considered. There was a high degree of constant features; the majority of variables in each model never flipped from their initialized value `F`, to `T`. Recall that a variable in the variational model is a feature in a feature model. Thus, for these datasets and for most variants, few features needed to be turned on in order for the feature model to be found satisfiabile. Additionally, this means these variational models are a compressed version of the corresponding set of plain models. Whether this ratio of constant-features generalizes to other software product lines is an open research question, however, this result implies that a variational model which only tracks the features which change value between variants, could greatly improve performance for subsequent queries on the variational model.

We believe variational models themselves are useful tools because they permit product analyses without a SAT solver. Figure 12 shows such a purely syntactic analysis: counting disjuncted clauses in the variational model as a representation of satisfiable plain models. These post-hoc analyses, may be useful to feature modelers as they direct attention to impactful versions of the feature model. For example, the change from V_7 to V_8 (128 to 256 Variants) of *fin* clearly constrained the feature model as the number of unsatisfiable models increased from 50% to 80%.

Data collection for all research questions required 7 days, 6 hours, and 21 minutes to complete. This experiment quadruples the run time of the conference version, because we repeat the experiment with four base solvers. Due to the amount of time required to generate the data, we limited the number of raw measurements to 3. Thus, each data point presented in our results is a bootstrapped average of 3 raw measurements. In contrast, the conference version of this work used 56 raw measurements per data point. Our results are inline with our former work even with the reduced sampling. We return to this point and discuss the reproducibility and standard error of measurement of our analysis in the threats to validity section.

Data	Alg.	2	4	8	16	32	64	128	256	512	1024
auto	$p \to p$	1.25	3.51	3.43	2.86						
auto	$v \to p$	0.96	3.26	3.24	2.62						
auto	$p \to v$	1.23	3.25	3.14	2.55						
auto	Mean	1.15	3.34	3.35	2.68						
fin	$p \to p$	3.43	4.13	4.11	3.57	3.39	3.03	2.79	2.61	2.39	2.11
fin	$v \to p$	3.40	4.20	4.07	3.51	3.35	3.01	2.77	2.61	2.55	2.16
fin	$p \to v$	3.39	3.98	3.86	3.32	2.90	2.40	2.19	2.01	1.81	1.58
fin	Mean	3.41	4.10	4.01	3.47	3.21	2.81	2.58	2.41	2.25	1.95

Table 1: Speedup by variant count, where $\texttt{SpeedUp} = \frac{\texttt{Algorithm}}{v \to v}$

RQ1: Execution Time as Variants Increase. Figure 13 and 14 show the total execution time of each algorithm as a function of variants using Z3 as the base solver. We see that the prototype solver ($v \to v$) shows reduced execution time compared to brute force ($p \to p$), the baseline case ($v \to p$), and the slowdown case ($p \to v$). We summarize Figure 13 and 14 in Table 1, which presents speedup by variant count. Speedup is the ratio of execution time

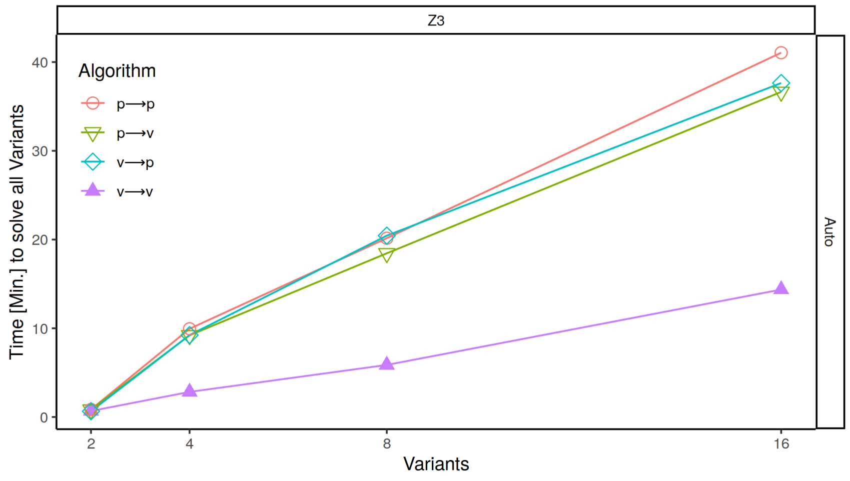


Fig. 13: (Auto) RQ1: performance as variants increase with Z3 base solver. $v \rightarrow v$ shows a speedup of 1.15–3.5x for the *auto* dataset depending on count of variants to solve.

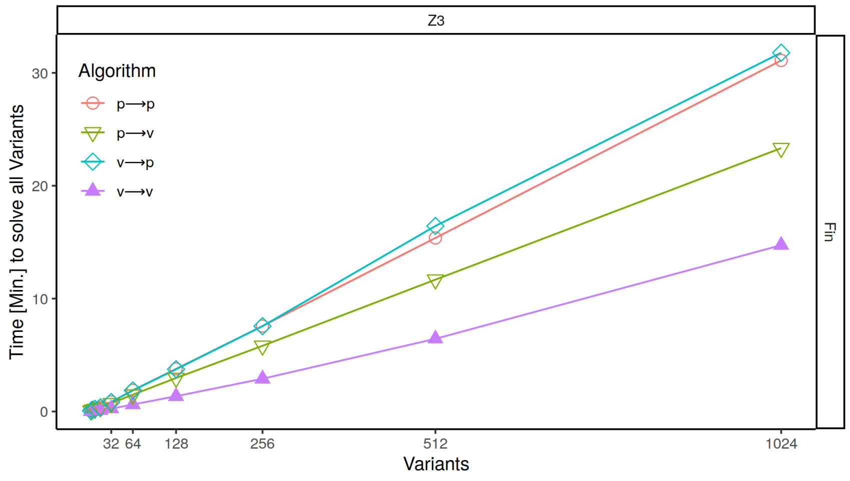


Fig. 14: (Financial) RQ1: performance as variants increase with Z3 base solver. $v \rightarrow v$ shows a speedup of 1.95–4.10x for the *fin* dataset depending on the count of variants to solve. Overlapping x-axis labels omitted.

Data	Alg.	2	4	8	16	32	64	128	256	512	1024
auto	$p \to p$	0.17	7.11	14.3	26.7						
auto	$v \to p$	-.03	6.40	14.6	23.3						
auto	$p \to v$	0.16	6.38	12.6	22.3						
auto	Mean	0.10	6.63	13.8	24.1						
fin	$p \to p$	0.05	0.12	0.18	0.27	0.60	1.26	2.42	4.67	8.93	16.4
fin	$v \to p$	0.05	0.12	0.17	0.27	0.59	1.25	2.39	4.66	10.0	17.1
fin	$p \to v$	0.05	0.11	0.16	0.25	0.48	0.87	1.61	2.92	5.24	8.60
fin	Mean	0.05	0.12	0.17	0.26	0.56	1.13	2.14	4.08	8.06	14.0

Table 2: Difference in execution time [min.] by solved variant count, where `Difference` = `Algorithm` − $v \to v$. A positive difference indicates $v \to v$'s execution time was reduced compared to the considered algorithm.

Data	Alg.	Model p-value	Adj. R^2	c estimate	c p-value	Intercept
auto	$p \to p$	5.04×10^{-11}	0.95	152.87	5.04×10^{-11}	-154.04
auto	$v \to p$	2.77×10^{-11}	0.96	150.97	2.77×10^{-11}	-146.60
auto	$v \to v$	3.33×10^{-08}	0.88	45.76	2.22×10^{-08}	-12.63
auto	$p \to v$	8.74×10^{-11}	0.95	136.84	8.74×10^{-11}	-125.37
fin	$p \to p$	2.20×10^{-16}	0.96	1.96	2.00×10^{-16}	-7.03
fin	$v \to p$	2.20×10^{-16}	0.97	2.01	2.00×10^{-16}	-7.73
fin	$v \to v$	2.20×10^{-16}	0.96	0.82	2.00×10^{-16}	-18.55
fin	$p \to v$	2.20×10^{-16}	0.95	1.49	2.00×10^{-16}	0.68

Table 3: Linear model comparison between algorithms. $v \to v$ shows reduced scaling factor estimate for both datasets compared to each other algorithm.

for each algorithm to $v \to v$. The "Mean" row is the average of all speedups for a given variant count. $v \to v$ demonstrates average speedups across all variants ranging from an average speedup of 1.15x (two variants) to 3.35x (eight variants). For *fin*, $v \to v$ shows average speedups ranging from 1.95x (1024 variants) to 4.10x (four variants). The only case where $v \to v$ shows a slowdown is the two variant case compared with $v \to p$ with a slowdown of 4%.

Recall that we hypothesized that as variant count increases, the more effective variational solving, and therefore $v \to v$, should be. We find that there is a peak speedup at the eight variant case for the *auto* dataset with an average speedup of 3.35x, and the four variant case for the *fin* dataset with an average speedup of 4.10x. After this peak, speedups monotonically decrease until the maximum variant cases are reached.

This observation is an artifact of the speedup metric. Table 2 displays the difference in execution time in minutes for each algorithm and variant count, where difference is the arithmetic difference in execution time between an algorithm and $v \to v$. When considering the raw difference in execution time, speedups are put into perspective. For example, the 1024 variant count of the *fin* dataset shows a 1.58x speedup of $v \to v$ compared to $p \to v$, while this speedup appears small it equates to *eight minutes* of wall time, which is more substantial than the 4.10x speedup that corresponds to *0.11 minutes* of wall time.

Table 3 displays the output of the linear model for both *auto* and *fin* datasets for each algorithm. Each linear model is statistically significant with a p-value of 2.2×10^{-16} for both datasets, furthermore we observe that the estimates for each algorithm are similarly significant. Table 3 directly answers RQ1; the constant scaling factor of $v \to v$ is reduced compared to each other algorithm for both datasets. For example, for the *auto* dataset, $v \to v$'s scaling factor is 45.76 compared to $v \to p$'s at 150.97. Therefore, we conclude that $v \to v$ scales more efficiently than brute force, the baseline case, and the slowdown case as variant count grows for both datasets. However, the $v \to v$ algorithm still grows exponentially, since the linear model is well fitted to both data sets and variant count (the x-axis) grows exponentially. Therefore, as variants to solve grow linearly, $v \to v$, like each other algorithm, grows exponentially but with a reduced constant factor.

The comparison between $p \to v$ and $v \to v$ is also notable. Recall that $p \to v$ performs accumulation/evaluation on a plain variant but *does not* generate and solve a variational core, which distinguishes it from $v \to v$. From Table 3, we see that $p \to v$ scales more efficiently than both $p \to p$ and $v \to p$. For example, for the *fin* dataset, $p \to v$ shows a scaling factor of 1.49 compared to 1.96 ($p \to p$) and 2.01 ($v \to p$). Thus, we conclude that accumulation/evaluation itself reduces execution time, yet most of the reported performance gains come from generating and solving the variational core.

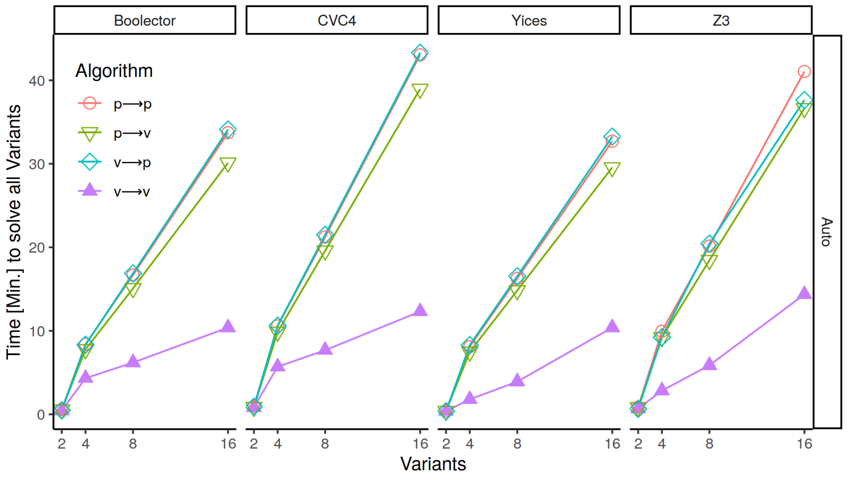


Fig. 15: (Auto) RQ2: performance as variants increase with each base solver. $v \to v$ shows a speedup of 2.8–3.5x for the *auto* dataset depending on base solver.

RQ2: Impact of the Base Solver on Execution Time. In the conference version of this work, we cited that all of results were dependent on Z3 as a threat

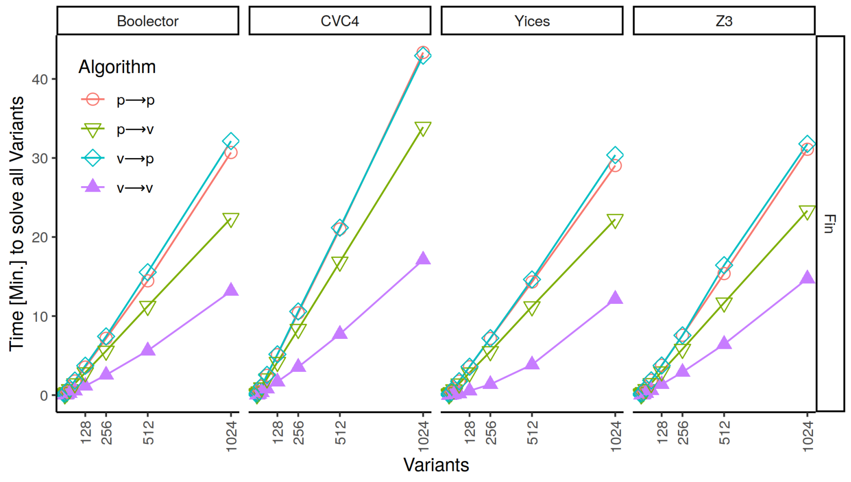


Fig. 16: (Financial) RQ2: performance as variants increase with each base solver. $v \to v$ shows a speedup of 2.16–2.51x for the *fin* dataset depending on base solver.

to validity. This research question addresses that threat and provides further insight into RQ1 by using other base solvers.

Figure 15 and 16 show total execution time as a function of variants for each algorithm and base solver, for both datasets, respectively. We find that the results for RQ1 are robust across every tested base solver. We summarize these results in Table 4 with Table 4a displaying the speedups and Table 4b presenting the execution time for the most variational case. For the *auto*

DataSet	Boolector	CVC4	Yices	Z3
auto	3.29	3.51	3.20	2.62
fin	2.44	2.51	2.50	2.16

(a) Speedup by solver for the most variational case; 16 variants for *auto*, 1024 for *fin*. Where `SpeedUp` $= \frac{v \to p}{v \to v}$

Boolector	CVC4	Yices	Z3
10.38	12.31	10.39	14.40
13.15	17.11	12.15	14.74

(b) Time [min.] to solve with $v \to v$ by solver.

Table 4: Time to solve and speedup of most variational case by solver.

dataset, $v \to v$ shows an average speedup of 2.60x across all variants. In the most variational case (16 variants), the greatest speedup was 3.5x with CVC4 as the base solver. The *fin* dataset shows an average speedup of 4.70x [14]. For the most variational case (1024 variants), CVC4 again showed the greatest speedup at 2.51x. These results are statistically significant: $v \to v$ is different from every other algorithm for each base solver with p-values of 2.77×10^{-4}

[14] Due to extreme outliers (10x–15.1x speedup) from Yices when solving 2–32 variants.

Data	2	4	8	16	32	64	128	256	512	1024
auto	7.26×10^{-4}	0.03	0.03	0.03						
fin	0.04	0.09	0.07	0.07	0.07	0.07	0.07	0.07	0.08	0.08

Table 5: P-values from Kruskal-Wallis test by variants and data set; a value ≤ 0.05 indicates the base solver had a statistically significant effect on execution time. Red values indicate insignificance.

$(v \to p)$, 1.06×10^{-2} $(p \to p)$, and 1.92×10^{-2} $(p \to v)$ for *auto* and 1.62×10^{-5} $(v \to p)$, 1.92×10^{-5} $(p \to p)$, and 1.70×10^{-4} $(p \to v)$ for *fin*.

We observe substantial differences in execution time between base solvers. For example, $v \to v$ with Boolector solved the 16 variant *auto* query formula in 10.38 minutes compared to 14.40 minutes with Z3. Yices was consistently the most performant base solver for all algorithms and all test cases. Yices demonstrated a high degree of speedup with a reduction of 3.97 minutes and 2.21 minutes in execution time, compared to the execution times reported with Z3. CVC4 is also noteworthy; CVC4 benefited the most from $v \to v$ for both datasets with a speedup 3.51x (*auto*) and 2.51x (*fin*).

Although, these results seem substantial, we fail to find a statistically significant effect from the base solver. Unfortunately, too much variance existed in the aggregate dataset, i.e., the data set that combines *auto* and *fin*, to assess an effect from the base solver. To reduce noise in the dataset, we performed an additional Kruskal-Wallis test grouping the data by variant count and dataset. Table 5 presents the p-values for this Kruskal-Wallis test. From this grouping, we find that the base solver had a statistically significant impact on execution time for the *auto* data set and the two-variant case of the *fin* dataset. Similarly, fitting a model to estimate the impact of the base solver on execution time, even with this additional grouping, failed. Our model explained no more variance in the dataset than random chance due to the limited data per data set and per variant count.

While we have not been able to show a statistically significant effect from the base solver with this dataset, such an effect may still exist but requires a more robust dataset to be detected. We nevertheless draw four conclusions from this experiment: First, Yices demonstrated the second best speedup with lowest execution times of all tested base solvers. We therefore find that Yices is an attractive target as the base solver for future variational SAT solvers. Second, some solvers are more sensitive to the variational solving algorithm than others. In particular, CVC4 demonstrated more sensitivity, and therefore speedup, compared to a less sensitive solver such as Z3. This implies that a base solver which shows poor performance within the typical use case ($v \to p$, i.e., an incremental context, and the solver is kept alive) may greatly benefit from the variational solving algorithm we have presented. Third, although the exact reasons behind this sensitivity are open research questions, these results imply that our use case (heavily exercising the incremental code paths) is peculiar and thus, selection of a base solver based on only its typical *non-incremental* performance may not be representative of its performance in the variational use case. Fourth, targeting the SMTLIB2 standard is a good implementation strategy for future variational SAT solvers. We found that the ability to try different base solvers on the same problems is convenient. In addition, targeting SMTLIB2 yields a modular design, allowing the variational solver the ability to painlessly add new SMT solvers rather than re-implementing the variational algorithm for each new base solver.

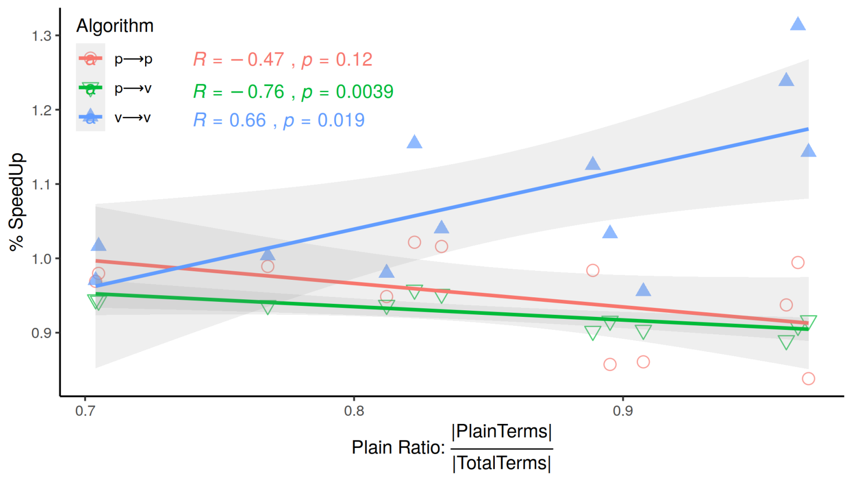


Fig. 17: RQ3: Performance as a function of plain ratio. We observe that sharing positively correlates to speedup only for $v \to v$, where `%` `SpeedUp` $= \frac{v \to p}{\text{Algorithm}}$.

RQ3: Impact of Sharing on Execution Time. We confirm the sharing hypothesis in Figure 17. Figure 17 is a scatter plot of speedup as a function of sharing ratio. Colored lines are linear models fitted to each algorithm, grey bands are the linear model's confidence intervals (95% confidence). Only $v \to v$ and $p \to p$ showed a statistically significant fit to a linear model. Furthermore, only $v \to v$ was found to be statistically different from $p \to p$ and $p \to v$, with p-values of 6.95×10^{-3} and 4.44×10^{-6} and *positively correlate* speedup with sharing ratio. We thus confirm that sharing positively correlates to speedups for the prototype variational solver for these datasets.

There are two sharing ratios of interest in the x-axis of Figure 17. The first is 0.73 since this is the point where the $v \to v$ linear model meets the $p \to p$ linear model. However, this point is an artifact of the analysis and plotting. The linear model for $p \to p$ resulted in a p-value of 0.12, indicating that the model did not predict the data better than chance, and therefore the model is a poor predictor. 0.70 is the second interesting point because this is the point where the models for $v \to v$ and $p \to v$ would meet had the data and correlations continued. Conceptually, this point indicates the

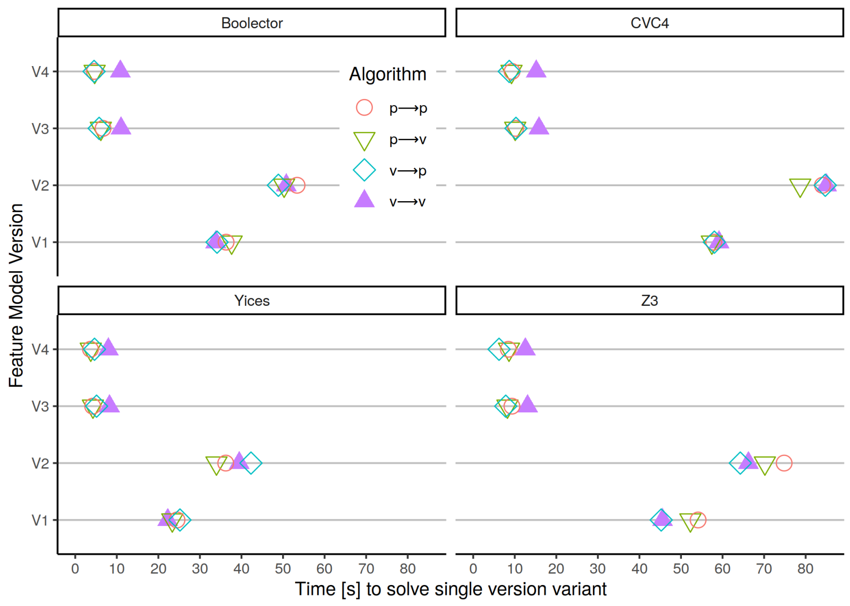


Fig. 18: (Auto) RQ4: Slowdown of $v \rightarrow v$ on plain formulas. We observe that $v \rightarrow v$ incurs an average slowdown of 9% for *auto*, when solving a version variant.

sharing ratio at which $v \rightarrow v$ begins to benefit from a high sharing ratio. Thus, the data is suggestive that low sharing ratios are detrimental to variational solving. More sharing ratio data of real-world VPL formulas is required to be conclusive. Unfortunately, to our knowledge, this is not a common metric in the incremental SAT community. Thus, an avenue for future work on variational SAT solving is to sample, aggregate, and report the distribution of real world VPL formulas and sharing ratios.

This data is evidence that a dataset's sharing ratio is an important factor in the performance of a variational SAT solver. Theoretically this makes sense; when the sharing ratio is high, the reduction engine produces a smaller variational core. With a smaller variational core, more reuse of plain terms occurs and thus computational time is saved in the base solver. Hence, another avenue of future work is to leverage the laws of the variational logic to automatically refactor input formulas to increase sharing. 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 product line versions or the representative 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.

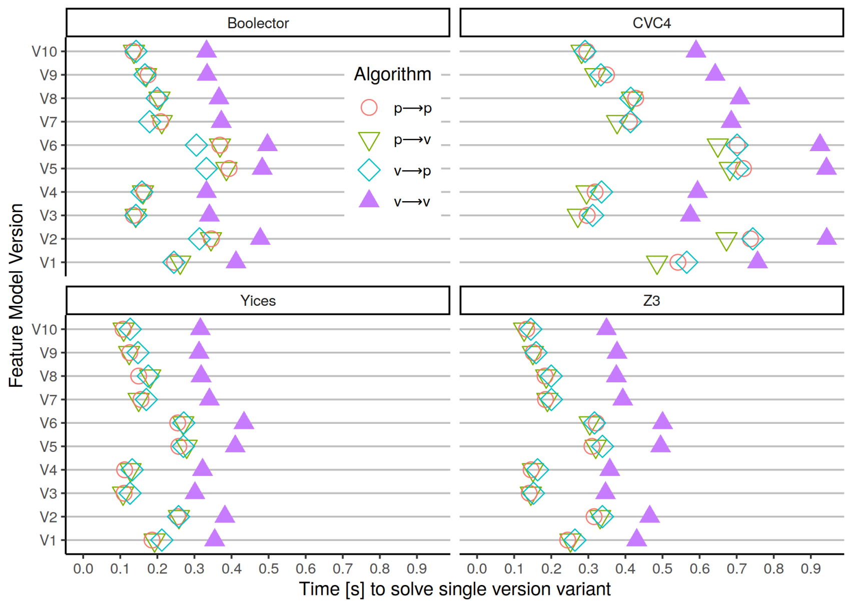


Fig. 19: (Financial) RQ4: Slowdown of $v \to v$ on plain formulas. We observe that $v \to v$ incurs an average slowdown of 75% for the *fin* dataset, when solving a version variant.

RQ4: Slowdown of VSAT on plain query formulas. Figure 18 and 19 display the bootstrapped averages of each version variant, for each algorithm, and base solver for the *auto*, and *fin* datasets, respectively. Given RQ3, and the sharing ratios of *fin*, we expect VSAT to show slowdowns for *fin*. This is observed in Figure 19 and is statistically significant for all versions. For *auto*, only the V_1 version variant showed a significant difference between the slowdown case, $p \to v$, and $v \to v$, and between the slowdown case $p \to v$, and the baseline case $v \to p$. Notably, $v \to v$ did not differ from the baseline case, $v \to p$. Graphically, Figure 18 is suggestive of a statistically significant difference between $v \to v$ and other algorithms as $v \to v$ is substantially different than other algorithms for the V_1 and V_2 cases for each base solver. However, Figure 18 does not show variance. With variance accounted for, the statistical tests conclude that the difference is not statistically significant, hence the discrepancy between the plot and our statistical results. That $p \to v$ was statistically different for V_1 suggests particular sets of plain SAT formulas (and thus a single VPL formula) may not respond well to the reduction engine, although the exact slowdown will be dependent on the SAT problem.

We conclude that our method of variational solving does show substantial slowdown outside of its intended use case, and is dependent on the query formula. While this result is unsurprising, in combination with RQ3 it further motivates future work to perform static analysis on the query formula

Data	Alg.	1	2	3	4
auto	$p \rightarrow p$	-1.18	151.69	304.55	457.42
auto	$v \rightarrow p$	3.37	154.33	305.30	456.27
auto	$v \rightarrow v$	33.14	78.90	124.66	170.42
auto	$p \rightarrow v$	11.48	148.32	285.16	422.0
fin	$p \rightarrow p$	-5.07	-3.11	-1.15	0.812
fin	$v \rightarrow p$	-5.72	-3.71	-1.70	0.31
fin	$v \rightarrow v$	-17.72	-16.90	-16.08	-15.26
fin	$p \rightarrow v$	2.17	3.66	5.15	6.64

Table 6: Linear model predications for one to four vaiants for both data sets and each algorithm. $v \rightarrow v$ shows an inflection point at one variant for *auto* and no variants for the *fin* data set.

during an initial parsing or construction phase, and then dispatch an appropriate solving algorithm. Thus, future variational solvers may be envisioned as a meta-SAT solver, i.e., it may choose to perform incremental solving, or variational solving, or a brute force depending on the query formula.

RQ5: Observing Inflection Point of Reduced Execution Time. Table 6 shows each algorithms' predicted execution time for one to four variants, using the models summarized in Table 3. We find that $v \rightarrow v$ only shows an inflection point at one variant for the *auto* dataset, with a predicted execution time of 33.14 seconds, compared to 3.37 seconds for the baseline ($v \rightarrow p$) and 11.48 seconds for the slowdown case ($p \rightarrow v$). We find no inflection point for the *fin* dataset. The predicted negative times in the $p \rightarrow p$ row for *auto* and most of the *fin* dataset indicate the breakdown of the model predictions. We fit a linear model because a linear model explained more variance in the data (i.e., had a higher adjusted R^2) than an exponential model. Yet, an exponential model is theoretically better suited since incremental SAT execution time is exponential in the number of SAT problems. Due to the linear model, the low variant count predictions for the *fin* dataset have less certainty, for example a 90% confidence interval around -16.08 (three variant case) of $v \rightarrow v$ ranges from -34.4 seconds to 0.6 seconds, i.e., an interval 2.1x as wide as the prediction. While the confidence interval of the same point in *auto* ranges from 21.90 to 135.9, and interval 1.44x the predicted value.

From this data, we draw two conclusions. First, inflection points in execution time exist but are heavily dependant on the dataset. Second, the inflection points are also dependent to the hardware. In contrast to the conference version of this work—which directly observed inflection points in the raw data at 4 variants for *auto* and 64 variants for *fin*—we find that the predicted inflection point is very low, at only a single variant for *auto*. We suspect that this discrepancy is due to convolved factors in the benchmarking system and hardware. This analysis was performed on a system with 32GB of RAM, and thus the benchmarking system, once running, never queried to disc. In contrast, the conference version benchmarking system had access to *only 500Mb* of RAM, therefore system operations such as paging were convolved and uncontrolled in that data hypothetically leading to an increase in constant time costs.

Understanding the inflection points is a high value item for future research. Future solvers could make direct use of them by analyzing a query formula to predict if a variational solver is likely to produce a speedup, if not then the solver could employ a standard incremental solver. Similarly, researchers could use the inflection points as a metric to compare variational solver implementations on the same dataset. Presumably, this would serve as a guiding light to allow researchers to test which implementation strategies scale and which do not. For example, if one solver demonstrates a lower inflection point on the same data, then that solver exhibits reuse at earlier variant count. Thus, by understanding the determining factors in the inflection point, and by using the inflection point as a metric, researchers gain a method with which to make progress in this domain.

Summary. We have found that the theoretical result of saving work by accumulation/evaluation, generation of a variational core, and solving the variational core translate to a reduction in execution time for two real-world data sets. From RQ1 we conclude that our approach is viable for some real-world workloads and that the majority of the reduction in execution time is derived from using variational cores. We hope that the concept of variational cores, as demonstrated in this work, can be useful to researchers in other variational domains.

Unfortunately, our data set lacked the statistical power to estimate and determine an effect on execution time from the base solver. However, our results are suggestive that the base solver has an impact, and that impact is particular to the specific base solver. Thus, RQ2 is inconclusive and left for future work when a more robust data set is in hand.

For RQ3 we sought to test the sharing hypothesis in the domain of variational SAT solvers. We have found strong evidence that the sharing hypothesis occurs in variational SAT solving. This result continues the accumulation of evidence that sharing ratios in variational systems are an important factor to the performance of these systems.

RQ4's purpose was to find and measure the overhead incurred by variational solving by solving plain formulas with VSAT. Such an overhead was observed and statistically meaningful for the *fin* dataset and one version of the *auto* dataset. If the constant time costs of variational SAT solving began to dominate execution time, then we would expect to observe a slowdown across *all* version variants. However this was not observed, that some version variants in the *auto* dataset did not show a slowdown suggests that the properties of a variant can have such a large impact on execution time that the overhead does not dominate. These results are suggestive of a variational phase change transition, the existence of which is an open question for future research.

In RQ5, we were primarily concerned with verifying the existence of an inflection point in execution time. We conclude that this inflection point does exist based on the models in this experiment and by direct observation in the conference version of this work. However, this leaves a more interesting problem: understanding the factors which produce the inflection point. In this work, we do not solve or address this problem, but believe that the threshold is dependent on numerous factors, such as the size and sharing ratio of query formula, the base solver, and the underlying hardware. Naturally, a path for future work is to identify these determining factors and model the variant threshold for a given query formula.

The results for RQ4 and RQ5 suggest that variational SAT solving should be used in conjunction with incremental SAT solving. For example, incremental SAT solvers could have a *variational engine* which is used when the input problem is well suited to variational SAT solving, completely invisible to the end-user. Or if the user inputs a VPL formula, then this solving engine would be the default method. Additionally, incremental solvers could employ a graduated approach, where the solver chooses between an engine which uses accumulation and evaluation, but does not use variational cores and a fully realized variational SAT solver.

Practicality. We have shown that variational SAT solving does reduce execution time when solving sets of related SAT problems. Unfortunately, our approach relies on encoding the set of SAT problems into a VPL formula. This is especially troubling in light of RQ3, which heavily implies that a poor encoding can have detrimental effects on the execution time of the variational solver, because the sharing ratio might be artificially low. Thus, a major and central question for future work is if the VPL formula can be automatically constructed in practice. Automated VPL formula construction is an open research question at this time, although there are several possible approaches we have identified. Since automated VPL construction is not the primary concern of this work, we leave it for future work and address several possible algorithms in Section 7.

It is important to note that this experiment only tests our methods on a small portion of real-world variational datasets. A variant count 10 times larger than that of *fin* is not only reasonable but also likely, and so we speculate what the performance of variational SAT solving will be with variant counts greater than 10,000 variants. This obviously requires further experimentation, but we hypothesize that the behavior is dependant on the sharing ratio of the VPL formula, the variant count, and the difficulty of the variant SAT problems.

If the sharing ratio is high, then based on RQ3 and RQ5 we would expect the speedup to continue to increase as the variants increase, because work is reused for each variant. In contrast, consider the case where the sharing ratio is zero or very low. In this case, we would expect the work done in generating and optimizing a variational core to never pay off, thus the solver would incur a slowdown due to the additional constant-time costs for each variant.

Similarly, the variant count and difficulty of solving each variant are important factors. There are four hypothetical scenarios, which are separated by answering the questions: “Is the execution time dominated by variational core generation and optimization" and "How much time do variational core optimizations have to pay off".

Given a variant count of n, a variational solver should only generate $\log_2 n$ variational cores. In the first scenario, n is low and the variants are easy to solve, such as in this experiment. In this scenario, there is not much time for the variational work to pay off, and the execution time is not dominated by the difficulty of each variant's SAT problem. Thus there is similarly not much time for the variational work itself. Instead, it is more likely that the variational work—generating and optimizing the variational cores—will become an important or even dominating factor in execution time. So it may not be worthwhile to do the variational work, instead it may be more efficient to directly solve the variants (as discussed in RQ5). Or, one might design a variational solver which detects this case and spends less time on variational core optimizations; since these optimizations have a lower probability to pay off. However, our results directly contradict this expectation; even when n is low and variants are easy to solve, generating variational cores still paid off for our test datasets (RQ1).

The second scenario is the most uncertain, n is low but the variant SAT problems are difficult. In this scenario, the cost of the variational work is low because execution time is likely to be dominated by solving the variant SAT problems. This provides more opportunity for the variational solver to perform aggressive optimizations. Although, there are not many variants for optimizations to pay off. We expect that any simplification to the variant SAT problem would produce a speedup because this is the dominating factor in execution time. However, this case is a balancing act, we need to do enough variational work to reduce the SAT time per variant, but we do not want to spend so much time on optimizations that the variational work becomes a dominating factor in execution time.

The third and fourth scenarios are similar; only now we assume that n is large which provides more time for variational work to pay off. In the third scenario, n is large but variant SAT problems are easy to solve. In this scenario, execution time should be dominated by generating the variational cores and driving the base solver to a variant to begin the SAT procedure. In contrast to the first scenario, generating and optimizing a variational core is still likely to pay off because even a small speedup per variant, when applied over many variants, will reduce total execution time. Similarly, optimizations should minimize the variational cores by increasing sharing. An increase in sharing should reduce the work required to drive the base solver to a variant because smaller variational cores require less traversal by accumulation, evaluation and choice removal. Therefore, we suspect that variational solving should still be performent in this scenario.

The fourth scenario is the best of both worlds; n is assumed to be large and the variants' SAT problems are assumed to be difficult. In this scenario the variational solver has enough time to generate and optimize the variational cores because execution time should be dominated by the variant's SAT problems. Similarly, this work has ample opportunity to pay off because the variant count is high. This is the crucial difference between this scenario and the second scenario; more time can be allocated to optimizations because the execution time is assumed to be dominated by the variant's SAT problem. Optimizations should improve the sharing ratio of the variational cores, but because the variant problems are difficult, any improvement to sharing may carry more information in the base solver forward to future variants, and produce a large reduction in work for the base solver. Thus, in this scenario, one could design a variational solver to detect this case and decide to more aggressively optimize because the payoff could be large, while the cost is small and likely non-dominating in execution time.

While our results are promising, variational SAT solving is not a general improvement over incremental SAT solving. Rather, in practice it should be viewed as a specialized improvement over incremental solving for classes of problems that satisfy two criteria. First, the set of SAT problems must be known in advance and have shared terms so that a VPL formula can be constructed to solve. Of course, one could use a variational SAT solver on a set of SAT problems which have no shared terms, but we would not expect variational solving to outperform incremental solving in such a case as shown in RQ4. Second, the number of SAT problems the user is interested in solving must be large. One may view variational SAT solvers as an optimizing compiler over incremental solvers. Thus, if the number of problems to solve is low, then the benefits of variational SAT solving are reduced (i.e., automating the solver interaction, optimizing the SMTLIB2 program, and automatically tracking the resulting models). Similarly, if the number of SAT problems is low then the speedup in execution time is less meaningful, even though one would no longer need to hand write the incremental programs. Therefore, one should view incremental SAT solving as a more general approach to solving sets of SAT problems, and variational SAT solving as a specialization of incremental SAT solving suited for use when the set of SAT problems is known, large, and consists of related problems.

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 (Nieke et al., 2018) 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 Sec. 5.1.4. Furthermore our results, and our prototype solver are based on the widely used Haskell library, sbv. While this is a likely to be a common implementation strategy for a variational solver (i.e., a solver built using a library rather than a foreign function interface, similar to tools built on top of Sat4j (Le Berre and Parrain, 2010)) it is nonetheless a threat to validity as our prototype directly depends on this library and its performance characteristics. To mitigate this threat we maintained the same version of sbv throughout the experiment, employed its interface to interoperate for each base solver, and enforced the same code paths through the library.

Due to the need for reduced measurements, our results are subject to the threat of validity incurred by random chance. It could have been the case that our results are merely statistical aberrations. We mitigate this threat in two ways. First, we aggressively employ statistical testing to detect statistically significant differences in the data which accounts for the number of comparisons and the size of the dataset. These tests are the reason RQ2 is not conclusive, even though our results are suggestive. Second, we measured the standard error of measurement by sampling the two variant and four variant case of $v \rightarrow v$ for the *fin* data set 56 times and *did not* include this data in Section 5.2. We found that the sampling distribution is tightly distributed about its mean. The two variant *fin* case showed a standard deviation of just 7.7 milliseconds with a mean of 4.096 seconds, and the four variant case showed a standard deviation of 29.59 milliseconds with a mean of 11.37 seconds, as reported by the gauge library. These results indicate that the reduced sampling of the distribution does still provide reasonably accurate data. In addition to this result, our results in this version of the work align with those from the conference version (Young et al., 2020), thereby increasing our confidence that the reduced sampling was not problematic.

We have concluded that inflection points exist in the execution time of a variational solver with RQ5. However, there exists is a threat to validity because our conclusion relies on the fitted linear model over the sample datasets and subsequent prediction of the inflection point rather than a direct observation in the raw data. We had expected to observe this inflection point directly in the raw data, as this was the case in the conference version of this paper. However, we believe the result is still valid due to the high quality of fit, the low measure of standard error of the linear models, and the direct observation of the inflection points on the same data in the conference version of this work. Taken as a whole, this increases our confidence in the result, although an experiment specifically designed for RQ5 would be more appropriate.

We have demonstrated 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, and it is possible that our encoding of Nieke et al.'s data was malformed. To mitigate the threat of VPL formula construction, we took several steps. We hand-crafted a custom parser to parse the set of SAT problems from Nieke et al.'s data. Using this parser, we parsed all of Nieke et al.'s data, checking for any failed SAT problems and ensuring that the number of successfully parsed problems matched the number of problems Nieke et al. reported. From the set of SAT problems, we created a VPL formula to represent the whole set in a post processing phase. Nieke et al.'s dataset is variational in its temporal ranges, so we post processed the set of SAT problems to create unique choices according to these temporal ranges, and then merged this set into a VPL formula. Then, we verified that the number of unique dimensions matched the number of feature model versions reported by Nieke et al. With the VPL formulas that represented all variants of a given data set, we projected smaller VPL formulas through configuration *and* restricted the variational solver with a variation context to ensure that the solver did not solve more variants than necessary. Lastly, during benchmarking, we statically analyzed each query formula to the solver *before* benchmarking, recording the number of choices, number of unique dimensions, the number of plain and variational terms, and the number of variants the formula represented. With this data we verified each benchmark was correctly benchmarking its respective VPL formula for each algorithm.

We do not provide a proof of the soundness of our methods. We mitigate this threat in several ways: We performed property-based testing (Claessen and Hughes, 2000) 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 $p \to v$, $v \to p$, and $p \to p$, an identical model p' was found by substituting p on the variational model returned from VSAT. We performed the property-based tests with 3,000 generated VPL formulas, finding no counter-examples.

6 Related Work

Similar Solvers, Related Techniques. Our work is most similar to Visser et al. (2012), 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 (Labs, 2020). 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 (Ataei et al., 2021a,b, 2017, 2018), graphics (Erwig and Smeltzer, 2018), data structures (Meng et al., 2017; Walkingshaw et al., 2014; Smeltzer and Erwig, 2017; Erwig et al., 2013), type systems (Campora III et al., 2018a,b; Chen et al., 2014b, 2012), error messages (Chen et al., 2017; Chen and Erwig, 2014; Chen et al., 2014a), variational execution systems (Chen et al., 2016; Wong et al., 2018; Meinicke, 2014) and now SAT solving. Our use of choices is similar to the concept of *facets* by Austin and Flanagan (2012) and *faceted execution* by Schmitz et al. (2018); Micinski et al. (2020); Austin et al. (2013), in that both choices and facets syntactically demarcate terms in an object language that must be specially handled, and yet must also operate with terms outside of the choice or facet. Facets have also been successfully applied to information-flow security and policy-agnostic programming. Our idea of representing variation in a non-traditional formula (a VPL formula in our case) is similar to the approach by Mauro (2021), which uses quantified boolean formulas to encode variation, and quantified boolean SAT solvers to detect anomalies in context-aware feature models. Notably, their approach has the benefit of avoiding incremental SAT solving altogether.

Our work is similar in spirit to incremental SAT solving itself, and we argue that one could view incremental SAT itself as a variational system. Thus, we provide a small literature review. First defined by Hooker (1993), incremental SAT was devised as a solution to verification and optimization problems in electronic design automation such as covering problems (Coudert and Madre, 1995), detecting delay faults (Kim et al., 2000a), and model checking (Clarke et al., 1986). The first incremental solver to gain traction was `SATIRE` by Whittemore et al. (2001). Eén and Sörensson (2004) made a major advance in incremental SAT with `MiniSat` by defining, documenting, and popularizing the implementation techniques required for an incremental SAT solver. `MiniSat` ((Eén and Sörensson, 2003) (Eén and Sörensson, 2004)) was the result of lessons learned from work on two other solver's called `SATZOO` and `SATNIK`. `MiniSat` simplified the existing notions of incrementality from the state of the art incremental solvers `SATIRE` and `PBS` (Aloul et al., 2002) and combined propagation strategies from the `Chaff` (Moskewicz et al., 2001) solver such as conflict-driven backtracking (Zhang et al., 2001) and dynamic variable ordering (Moskewicz et al., 2001). These combinations lead to a solver that was performant, and whose implementation was small and communicative. That same year, the first SMTLIB standard would be proposed by Tinelli (2003) although incremental SAT commands would not be incorporated until the 2.0 version (Barrett et al., 2010) in 2010, with the addition of an assertion stack developed by Kim et al. (2000b).

Additionally, one could conceive of variational SAT solving as an satisfiability-modulo theories (SMT) background theory (Barrett and Tinelli, 2018; Biere et al., 2009) that could be included in SMT solvers, rather than as a system which includes a SAT/SMT solver as a black box reasoning engine. This approach is feasible but is only a difference of implementation strategy, and we expect that a product ready variational SAT solving would be implemented this way. Since, the primary goal of this work was to explore, formalize, and prototype variational SAT solving, we chose to design the variation solver as a set of rule based transformations over a new logic, that offloads SAT problems to a reasoning engine. This has many benefits. First, it creates a clean separation between the variational solver and the SAT/SMT solver. From a software engineering perspective, this separation increases the cohesion of each sub-system and limits the coupling of the variational solver to the base solver. Second, because our design has strong separation of concerns, our prototype can replace base solvers or even use several different base solvers at the same time, for different variants; such a capability would not be possible without this design. Similarly, strong separation of concerns allows us to experiment with parallel and concurrent variational solving, where variants are solved by different base solver instances, at the same time. Third, our design does not disallow implementing a variational SAT solver as an SMT theory. Rather, we believe that translation to an SMT theory, based on our formalization, is straightforward precisely because we have formalized variational SAT solving as a set of rule based transformations over a new logic. Thus, it is reasonable that a future variational SAT solver *is* implemented as a SMT theory just as we have suggested in Section 5.2.

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 (Batory, 2005; Czarnecki and Wąsowski, 2007; Mendonça et al., 2008) for analysis purposes (Benavides et al., 2010; Thüm et al., 2014; Galindo et al., 2019). Many analyses have been implemented using SAT solving such as Thüm et al. (2014), including feature-model analysis (Benavides et al., 2010; Galindo et al., 2019), parsing (Kästner et al., 2011), dead-code analysis (Tartler et al., 2011), code simplification (von Rhein et al., 2015), type checking (Thaker et al., 2007), consistency checking (Czarnecki and Pietroszek, 2006), dataflow analysis (Liebig et al., 2013), model checking (Classen et al., 2013), variability-aware execution (Nguyen et al., 2014), testing (Carmo Machado et al., 2014), product sampling (Medeiros et al., 2016; Varshosaz et al., 2018), product configuration (Sayyad et al., 2013), optimization of non-functional properties (Siegmund et al., 2012), and variant-preserving refactorings (Fenske et al., 2017). 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 (Visser et al., 2012), and especially in *speculative* program analyses that involve generating and exploring huge numbers of variations of a program, for example, as in counterfactual (Chen and Erwig, 2014) and migrational (Campora III et al., 2018b,a) typing. Furthermore, we believe that variational solving could provide a basis for similar 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. (2018) 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 (Thüm et al., 2014), 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 (Apel et al., 2010; Huang et al., 2011). 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 SAT problems by delaying consideration of the feature model until after the analysis and only using it to rule out false positives (Bodden et al., 2013; Classen et al., 2013; Liebig et al., 2013), a technique known as late feature-model consideration (Thüm et al., 2014). Bodden et al. (2013) found that this technique increases the overall efficiency of static analysis (Bodden et al., 2013), while Classen et al. (2013) 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 (Acher et al., 2011; Krieter et al., 2016) or decomposition (Schröter et al., 2016). 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 (Krieter et al., 2018), 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. (2018) found for configuration processes.

7 Conclusion and Future Work

Variational SAT solving offers numerous advantages over current methods. Variational models, as solutions to variational SAT 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.

We have demonstrated that sharing is an important factor in variational SAT solving. While the magnitude of its effect is unknown, our analysis forms a foundation for future research. For feature modelers, variational SAT solving offers the practical benefits of a faster and more flexible analysis tool. Outside the domain of software product lines, variational SAT solving provides a framework and logic that directly represents variation irrespective of the application domain, thus providing a new method to study variation itself.

There are several avenues of future work. In this section we explore two of the promising directions: (1) extending the formalism to SMT solving and (2) automating the construction of VPL formulas.

We view extending the formalism and implementation to SMT solving and automatic VPL formula construction as the highest value direction of future work. Extending to SMT solving is straightforward, and is complete at the time of writing!Young (2021). In our prototype variational SMT solver, VPL is extended to consider numeric expressions. Numeric expression are introduced to the variational formula through new inequalities such as $\leq$. Then appropriate cases for each new relation are added to accumulation, evaluation and choice removal. The new cases include new rules over arithmetic relations such as, $+$, $-$, $*$ and inequalities, but the semantics of accumulation, evaluation and choice removal are exactly the same as presented above. Variational models are also extended to contain values other than Booleans. The models use the *ite* (if-then-else) function from the SMTLIB2 standard to create a sequence of heterogeneous values. The values are selected for by the if-condition, which is the variant context for the value in the then-case. The else-case becomes a pointer to the rest of the sequence, and the final else-case is the initialized value for each variable. With this strategy, we achieved a variational SMT solver extended with all SMTLIB2 standardized theories.

Incrementally and automatically constructing VPL formulas is the second major avenue of future work. The requirement to provide a VPL formula creates a high barrier to entry for end-users. Thus, if the VPL formula can be automatically constructed then usability of the tool increases. We believe that constructing a VPL formula as new variants occur is possible in theory (as described in Section 3). However the problem is far from trivial. From RQ3, we observed that the sharing ratio was a significant factor in performance. Thus, the problem is not simply devising an algorithm which inputs a set of C_2 formulas and produces a VPL formula. Rather, the solution must deliver a VPL formula *which also* maximizes sharing!

The problem of incrementally building a VPL formula reduces to taking two candidate VPL formulas (each of which could be plain) as input, and returning a fitness metric; where a high result implies a high degree of sharing between the candidates. With this fitness metric, well suited pairs of combination could be constructed to maximize sharing. One could then imagine a top down approach which combines two formulas element by element wrapping non-shared sub-terms in a unique choice, and then apply the rules in Section 3 to increase the sharing ratio.

However there are two considerations. The first consideration is the computational complexity of finding the fitness metric between two arbitrary VPL formulas. There are several possible algorithms, such as string edit distance, and graph edit distance over the abstract syntax tress of both candidate formulas. String comparison algorithms such as Levenshtein distance (Levenshtein, 1966) or Hamming distance (Hamming, 1950) are promising as both have implementations which run in polynomial time. Graph edit distance is more straightforward, and would not require serialization of the formulas to strings.

However, graph edit distance is NP-Complete with an approximate solution that is APX-hard (Lin, 1994). Although the computational complexity is high, most graph edit distance algorithms work well in practice, and it is likely that our use case is simpler than the worse case, such as the enormous graphs found in social networks. Similarly, it is likely that this domain would significantly benefit from heuristics such as longest common sub-string because VPL with a long shared segment of terms are likely to have other shared terms separated by choices.

The second consideration is the number of comparisons to make with the fitness calculating algorithm. To give some insight into the problem, given a set of n SAT problems, we want to create a new set of $\lceil \frac{n}{2} \rceil$ pairs, combine each pair into a VPL formula with unique choices, and recursively process the new set until we arrive at the base case of a single VPL formula. But this means that there are $\frac{n!}{2(n-2)!}$ comparisons to make at each recursive step. Thus, a single iteration of this algorithm with an initial set of 10 formulas would incur 45 comparisons, 100 problems would incur 4,950 comparisons, and 1,000 problems would incur 499,500 comparisons. Furthermore, we can calculate the total count of comparisons required to arrive at the base case by summing the number of comparisons at each recursive level. For an initial set of 10 problems, the algorithm would make 45 ($\frac{10!}{2(8)!}$) in its first step, then 10 ($\frac{5!}{2(3)!}$) and finally three 3 ($\frac{3!}{2!}$) comparisons to reduce the inital set to a single VPL formula. While this process incurs substantial computational complexity, these upper bounds are likely to be reduced using approximation algorithms or by sorting the set before processing, although the effect of each is left to future work. Fundamentally, this problem is a search problem with the fitness metric as a heuristic. Thus, we believe significant progress could be made in automating the construction of VPL formulas by using high performance heuristic-based search algorithms such as A*, beam search, or Alpha-beta pruning (Russell and Norvig, 2009).

Both directions of future work require more data for an evaluation. For variational SMT solving, we require more data sets to repeat the analysis *for each* theory. Interactions between variation and each SMT theory are an open question, and it could be the case that the interaction between variation and a particular theory (e.g., fixed size bit vector theory) produces harder SMT problems than its plain counterpart. For algorithms which automatically construct VPL formulas, we require a set of realistic SAT formulas, and by extension SMT formulas. Accumulating this data is feasible; our only constraints are that the set of SAT problems are not synthetic, and that there is some sharing between the problems in the set. Ideally, the amount of sharing would range from no sharing, to full sharing in the SAT problems. Such a data set would provide valuable information on the distribution of sharing ratios in real-world problem sets and would give insight into variational SAT performance at a lower sharing ratio than the minimum found in this work (0.7). Thus, the immediate goal of our future work is to accumulate, record, and make available a robust dataset of related SAT/SMT problems.

8 Declarations

This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1).

9 Acknowledgments

We gratefully acknowledge discussions on variational solving with Michael Nieke and attendees of FOSD 2018. We thank colleagues without whom this work would not be possible: Parisa Ataei for help with choice calculus, David Thrane Christiansen for the inspiration behind our approach, and Levent Erkok for his support with the implementation.

References

Acher M, Collet P, Lahire P, France RB (2011) Slicing Feature Models. In: Proc. Int'l Conf. on Automated Software Engineering (ASE), IEEE, pp 424–427

Aloul FA, Ramani A, Markov IL, Sakallah KA (2002) Generic ilp versus specialized 0-1 ilp: An update. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, Association for Computing Machinery, New York, NY, USA, ICCAD '02, p 450–457, DOI 10.1145/774572.774638, URL `https://doi.org/10.1145/774572.774638`

Ananieva S, Kowal M, Thüm T, Schaefer I (2016) Implicit Constraints in Partial Feature Models. In: Int. Work. on Feature-Oriented Software Development (FOSD), ACM, pp 18–27

Apel S, Scholz W, Lengauer C, Kästner C (2010) Language-Independent Reference Checking in Software Product Lines. In: Int. Work. on Feature-Oriented Software Development (FOSD), ACM, pp 65–71

Ataei P, Termehchy A, Walkingshaw E (2017) Variational Databases. In: Int. Symp. on Database Programming Languages (DBPL), ACM, pp 11:1–11:4

Ataei P, Termehchy A, Walkingshaw E (2018) Managing Structurally Heterogeneous Databases in Software Product Lines. In: VLDB Workshop: Polystores and Other Systems for Heterogeneous Data (Poly)

Ataei P, Khan F, Walkingshaw E (2021a) A Variational Database Management System. In: ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE), to appear

Ataei P, Li Q, Walkingshaw E (2021b) Should Variation Be Encoded Explicitly in Databases? In: Int. Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS), pp 3:1–3:9

Austin TH, Flanagan C (2012) Multiple facets for dynamic information flow. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp 165–178

Austin TH, Yang J, Flanagan C, Solar-Lezama A (2013) Faceted execution of policy-agnostic programs. In: Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Association for Computing Machinery, New York, NY, USA, PLAS '13, p 15–26, DOI 10.1145/2465106.2465121, URL `https://doi.org/10.1145/2465106.2465121`

Balyo T, Froleyks N, Heule M, Iser M, Järvisalo M, Suda M (eds) (2020) Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, Department of Computer Science, University of Helsinki, Finland

Barrett C, Tinelli C (2018) Satisfiability modulo theories. In: Handbook of Model Checking, Springer, pp 305–343

Barrett C, Stump A, Tinelli C (2010) The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa, available at `www.SMT-LIB.org`

Barrett C, Conway CL, Deters M, Hadarean L, Jovanović D, King T, Reynolds A, Tinelli C (2011) Cvc4. In: Gopalakrishnan G, Qadeer S (eds) Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 171–177

Barrett C, Fontaine P, Tinelli C (2016) The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`

Batory D (2005) Feature Models, Grammars, and Propositional Formulas. In: ACM SIGSOFT Int. Systems and Software Product Line Conf. (SPLC), Springer, pp 7–20

Benavides D, Ruiz-Cortés A, Trinidad P (2005) Automated Reasoning on Feature Models. In: Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE), pp 491–503

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems 35(6):615–708

Biere A, Biere A, Heule M, van Maaren H, Walsh T (2009) Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press

Bittner PM, Thüm T, Schaefer I (2019) Sat encodings of the at-most-k constraint. In: Ölveczky PC, Salaün G (eds) Software Engineering and Formal Methods, Springer International Publishing, Cham, pp 127–144

Bodden E, Tolêdo T, Ribeiro M, Brabrand C, Borba P, Mezini M (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, pp 355–364

Brummayer R, Biere A (2009) Boolector: An efficient SMT solver for bitvectors and arrays. In: Kowalewski S, Philippou A (eds) Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, Springer, Lecture Notes in Computer Science, vol 5505, pp 174–177, URL https://doi.org/10.1007/978-3-642-00768-2_16

Brummayer R, Biere A (2021) Boolector: A Satisfiability Modulo Theories (SMT) solver for the theories of fixed-size bit-vectors, arrays and uninterpreted functions. https://boolector.github.io/, accessed at Sept 12, 2021

Campora III JP, Chen S, Erwig M, Walkingshaw E (2018a) Migrating Gradual Types. Proc of the ACM on Programming Languages (PACMPL) issue ACM SIGPLAN Symp on Principles of Programming Languages (POPL) 2:15:1–15:29

Campora III JP, Chen S, Walkingshaw E (2018b) 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) 2:98:1–98:30

Carmo Machado ID, McGregor JD, Cavalcanti YaC, De Almeida ES (2014) On Strategies for Testing Software Product Lines: A Systematic Literature Review. J Information and Software Technology (IST) 56(10):1183–1199

Chen S, Erwig M (2014) Counter-Factual Typing for Debugging Type Errors. In: ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp 583–594

Chen S, Erwig M, Walkingshaw E (2012) An Error-Tolerant Type System for Variational Lambda Calculus. In: ACM SIGPLAN Int. Conf. on Functional Programming (ICFP), pp 29–40

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

Chen S, Erwig M, Walkingshaw E (2014b) Extending Type Inference to Variational Programs. ACM Trans on Programming Languages and Systems 36(1):1:1–1:54

Chen S, Erwig M, Walkingshaw E (2016) A Calculus for Variational Programming. In: European Conf. on Object-Oriented Programming (ECOOP), LIPIcs, vol 56, pp 6:1–6:26

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

Claessen K, Hughes J (2000) Quickcheck: A lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, Association for Computing Machinery, New York, NY, USA, ICFP '00, p 268–279, DOI 10.1145/351240.351266, URL https://doi.org/10.1145/351240.351266

Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263, DOI 10.1145/5397.5399, URL http://doi.acm.org/10.1145/5397.5399

Classen A, Cordy M, Schobbens PY, Heymans P, Legay A, Raskin JF (2013) Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. IEEE

Trans on Software Engineering 39(8):1069–1089

Coudert O, Madre JC (1995) New ideas for solving covering problems. In: Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference, Association for Computing Machinery, New York, NY, USA, DAC '95, p 641–646, DOI 10.1145/217474.217603, URL `https://doi.org/10.1145/217474.217603`

Czarnecki K, Pietroszek K (2006) Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In: ACM SIGPLAN Conf. on Generative Programming and Component Engineering, ACM, pp 211–220

Czarnecki K, Wąsowski A (2007) Feature Diagrams and Logics: There and Back Again. In: ACM SIGSOFT Int. Systems and Software Product Line Conf. (SPLC), IEEE, pp 23–34

Dutertre B (2014) Yices 2.2. In: Biere A, Bloem R (eds) Computer-Aided Verification (CAV'2014), Springer, Lecture Notes in Computer Science, vol 8559, pp 737–744

Eén N, Sörensson N (2003) Temporal induction by incremental sat solving. Electronic Notes in Theoretical Computer Science 89(4):543–560

Eén N, Sörensson N (2004) An extensible sat-solver. In: Giunchiglia E, Tacchella A (eds) Theory and Applications of Satisfiability Testing, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 502–518

Erkok L (2011) SBV: SMT Based Verification: Symbolic Haskell theorem prover using SMT solving. Website, available online at `https://hackage.haskell.org/package/sbv-8.10`; visited on Feb 14th, 2020.

Erwig M, Smeltzer K (2018) Variational Pictures. In: Int. Conf. on the Theory and Application of Diagrams, LNAI 10871, pp 55–70

Erwig M, Walkingshaw E (2011) The Choice Calculus: A Representation for Software Variation. ACM Trans on Software Engineering and Methodology (TOSEM) 21(1):6:1–6:27

Erwig M, Walkingshaw E, Chen S (2013) An Abstract Representation of Variational Graphs. In: Int. Work. on Feature-Oriented Software Development (FOSD), ACM, pp 25–32

Fenske W, Meinicke J, Schulze S, Schulze S, Saake G (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, pp 316–326

Galindo JA, Benavides D, Trinidad P, Gutiérrez-Fernández AM, Ruiz-Cortés A (2019) Automated Analysis of Feature Models: Quo Vadis? Computing 101(5):387–433

Garson J (2018) Modal logic. In: Zalta EN (ed) The Stanford Encyclopedia of Philosophy, fall 2018 edn, Metaphysics Research Lab, Stanford University

Gent IP, Walsh T (1994) The sat phase transition. In: In Proc. ECAI-94, pp 105–109

Hamming RW (1950) Error detecting and error correcting codes. Bell System technical journal 29(2):147–160

Harper R (2016) Practical Foundations for Programming Languages. Cambridge University Press

Holm S (1979) A simple sequentially rejective multiple test procedure. Scandinavian Journal of Statistics 6(2):65–70, URL http://www.jstor.org/stable/4615733

Hooker JN (1993) Solving the incremental satisfiability problem. DOI 10.1184/R1/6708044.v1, URL https://kilthub.cmu.edu/articles/journal_contribution/Solving_the_Incremental_Satisfiability_Problem/6708044/1

Huang SS, Zook D, Smaragdakis Y (2011) Statically Safe Program Generation with SafeGen. Science of Computer Programming (SCP) 76(5):376–391

Hudak P, Peyton Jones S, Wadler P, Boutel B, Fairbairn J, Fasel J, Guzmán MM, Hammond K, Hughes J, Johnsson T, Kieburtz D, Nikhil R, Partain W, Peterson J (1992) Report on the programming language haskell: A non-strict, purely functional language version 1.2. SIGPLAN Not 27(5):1–164, DOI 10.1145/130697.130699, URL http://doi.acm.org/10.1145/130697.130699

Huet G (1997) The zipper. Journal of Functional Programming 7(5):549–554, DOI 10.1017/S0956796897002864

Kästner C, Giarrusso PG, Rendel T, Erdweg S, Ostermann K, Berger T (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, pp 805–824

Kästner C, Ostermann K, Erdweg S (2012) A Variability-Aware Module System. In: Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), ACM, pp 773–792

Kim J, Whittemore J, Marques-Silva JaP, Sakallah K (2000a) On applying incremental satisfiability to delay fault testing. In: Proceedings of the Conference on Design, Automation and Test in Europe, Association for Computing Machinery, New York, NY, USA, DATE '00, p 380–384, DOI 10.1145/343647.343801, URL https://doi.org/10.1145/343647.343801

Kim J, Whittemore JP, Marques-Silva J, Sakallah KA (2000b) On Solving Stack-Based Incremental Satisfiability Problems. In: Proc. of IEEE International Conference on Computer Design (ICCD), Austin, Texas, pp 379–382

Kleene SC (1968) Introduction to metamathematics. Ishi Press

Kocher P, Horn J, Fogh A, , Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2019) Spectre attacks: Exploiting speculative execution. In: 40th IEEE Symposium on Security and Privacy (S&P'19)

Kowal M, Ananieva S, Thüm T (2016) Explaining Anomalies in Feature Models. In: Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE), ACM, pp 132–143

Krieter S, Schröter R, Thüm T, Fenske W, Saake G (2016) Comparing Algorithms for Efficient Feature-Model Slicing. In: ACM SIGSOFT Int. Systems and Software Product Line Conf. (SPLC), ACM, pp 60–64

Krieter S, Thüm T, Schulze S, Schröter R, Saake G (2018) Propagating Configuration Decisions with Modal Implication Graphs. In: Proc. Int'l Conf. on Software Engineering (ICSE), ACM, pp 898–909

Kästner C, Giarrusso PG, Ostermann K (2011) Partial preprocessing c code for variability analysis. In: In Proc. 5th ACM Workshop on Variability Modeling of Software-Intensive Systems, pp 127–136

Labs R (2020) Redis. `https://redis.io/`, accessed at May 4th, 2020

Larabel M (2020) 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`, accessed at March 25th, 2020

Le Berre D, Parrain A (2010) The Sat4j Library, Release 2.2 7(2-3):59–64

Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8):707–710, doklady Akademii Nauk SSSR, V163 No4 845-848 1965

Liebig J, Kästner C, Apel S (2011) Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: Int. Conf. on Aspect-Oriented Software Development, pp 191–202

Liebig J, von Rhein A, Kästner C, Apel S, Dörre J, Lengauer C (2013) Scalable Analysis of Variable Software. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), ACM, pp 81–91

Lin CL (1994) Hardness of approximating graph transformation problem. In: Du DZ, Zhang XS (eds) Algorithms and Computation, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 74–82

Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M (2018) Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18)

Mauro J (2021) Anomaly detection in context-aware feature models. In: 15th International Working Conference on Variability Modelling of Software-Intensive Systems, Association for Computing Machinery, New York, NY, USA, VaMoS'21, DOI 10.1145/3442391.3442405, URL `https://doi.org/10.1145/3442391.3442405`

Mauro J, Nieke M, Seidl C, Yu IC (2017) Anomaly Detection and Explanation in Context-Aware Software Product Lines. In: ACM SIGSOFT Int. Systems and Software Product Line Conf. (SPLC), ACM, pp 18–21

Medeiros F, Kästner C, Ribeiro M, Gheyi R, Apel S (2016) A Comparison of 10 Sampling Algorithms for Configurable Systems. In: Proc. Int'l Conf. on Software Engineering (ICSE), ACM, pp 643–654

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

Meinicke J (2019) Variational debugging: Understanding differences among executions. PhD dissertation, University of Magdeburg

Mendonça M, Wąsowski A, Czarnecki K, Cowan D (2008) Efficient Compilation Techniques for Large Scale Feature Models. In: ACM SIGPLAN Conf. on Generative Programming and Component Engineering, ACM, pp 13–22

Meng M, Meinicke J, Wong CP, Walkingshaw E, Kästner C (2017) A Choice of Variational Stacks: Exploring Variational Data Structures. In: Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS), ACM, pp

28–35

Micinski K, Darais D, Gilray T (2020) Abstracting faceted execution. In: 2020 IEEE 33rd Computer Security Foundations Symposium (CSF), pp 184–198, DOI 10.1109/CSF49147.2020.00021

Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th Annual Design Automation Conference, ACM, New York, NY, USA, DAC '01, pp 530–535, DOI 10.1145/378239.379017, URL `http://doi.acm.org/10.1145/378239.379017`

de Moura L, Bjørner N (2008) Z3: An efficient smt solver. In: Ramakrishnan CR, Rehof J (eds) Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 337–340

Nadel A, Ryvchin V, Strichman O (2014) Ultimately incremental sat. In: Sinz C, Egly U (eds) Theory and Applications of Satisfiability Testing – SAT 2014, Springer International Publishing, Cham, pp 206–218

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

Nguyen HV, Kästner C, Nguyen TN (2014) Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In: Proc. Int'l Conf. on Software Engineering (ICSE), ACM, pp 907–918

Nieke M, Mauro J, Seidl C, Thüm T, Yu IC, Franzke F (2018) Anomaly Analyses for Feature-Model Evolution. In: ACM SIGPLAN Conf. on Generative Programming and Component Engineering, ACM, pp 188–201

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

R Core Team (2020) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL `https://www.R-project.org/`

Rescher N (1969) Many-Valued Logic. New York: Mcgraw-Hill

von Rhein A, Grebhahn A, Apel S, Siegmund N, Beyer D, Berger T (2015) Presence-Condition Simplification in Highly Configurable Systems. In: Proc. Int'l Conf. on Software Engineering (ICSE), IEEE, pp 178–188

Russell S, Norvig P (2009) Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall Press, USA

SAT Competition (2021) SAT Competition 2021: Incremental Library Track. `https://satcompetition.github.io/2021/track_incremental.html`, accessed at May 21, 2022

Sayyad AS, Ingram J, Menzies T, Ammar H (2013) Scalable product line configuration: A straw to break the camel's back. In: Proc. Int'l Conf. on Automated Software Engineering (ASE), IEEE, pp 465–474

Schmitz T, Algehed M, Flanagan C, Russo A (2018) Faceted secure multi execution. In: CCS '18

Schröter R, Krieter S, Thüm T, Benduhn F, Saake G (2016) Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable

Systems. In: Proc. Int'l Conf. on Software Engineering (ICSE), ACM, pp 667–678

Siegmund N, Rosenmüller M, Kuhlemann M, Kästner C, Apel S, Saake G (2012) SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. Software Quality Journal (SQJ) 20(3-4):487–517

Smeltzer K, Erwig M (2017) Variational Lists: Comparisons and Design Guidelines. In: ACM SIGPLAN Int. Workshop on Feature-Oriented Software Development, pp 31–40

Tartler R, Lohmann D, Sincero J, Schröder-Preikschat W (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, pp 47–60

Thaker S, Batory D, Kitchin D, Cook W (2007) Safe Composition of Product Lines. In: ACM SIGPLAN Conf. on Generative Programming and Component Engineering, ACM, pp 95–104

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

Tinelli C (2003) The smt-lib format: an initial proposal. URL `https://smtlib.cs.uiowa.edu/papers/pdpar-proposal.pdf`, accessed September 28, 2021

Varshosaz M, Al-Hajjaji M, Thüm T, Runge T, Mousavi MR, Schaefer I (2018) A Classification of Product Sampling for Software Product Lines. In: ACM SIGSOFT Int. Systems and Software Product Line Conf. (SPLC), ACM, pp 1–13

Visser W, Geldenhuys J, Dwyer MB (2012) Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In: Proc. Int'l Symposium on Foundations of Software Engineering (FSE), ACM, pp 58:1–58:11

Walkingshaw E (2013) The Choice Calculus: A Formal Language of Variation. PhD thesis, Oregon State University, `http://hdl.handle.net/1957/40652`

Walkingshaw E, Kästner C, Erwig M, Apel S, Bodden E (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!), pp 213–226

Whittemore J, Kim J, Sakallah K (2001) Satire: A new incremental satisfiability engine. In: Proceedings of the 38th Annual Design Automation Conference, Association for Computing Machinery, New York, NY, USA, DAC '01, p 542–545, DOI 10.1145/378239.379019, URL `https://doi.org/10.1145/378239.379019`

Wilcoxon F (1945) Individual comparisons by ranking methods. Biometrics Bulletin 1(6):80–83, URL `http://www.jstor.org/stable/3001968`

Wong CP (2021) Beyond configurable systems: Applying variational execution to tackle large search spaces. PhD dissertation, Carnegie Mellon University

Wong CP, Meinicke J, Lazarek L, Kästner C (2018) Faster variational execution with transparent bytecode transformation. Proc ACM Program Lang 2(OOPSLA), DOI 10.1145/3276487, URL `https://doi.org/10.`

1145/3276487

Young J (2021) Variational Satisfiability Solving. PhD thesis, Oregon State University, https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/dv140182g?locale=en

Young JM, Walkingshaw E, Thüm T (2020) Variational satisfiability solving. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A, Association for Computing Machinery, New York, NY, USA, SPLC '20, DOI 10.1145/3382025.3414965, URL https://doi.org/10.1145/3382025.3414965

Zhang L, Madigan CF, Moskewicz MH, Malik S (2001) Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design, IEEE Press, Piscataway, NJ, USA, ICCAD '01, pp 279–285, URL http://dl.acm.org/citation.cfm?id=603095.603153