Variational Satisfiability Solving

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

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

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

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

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

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

\[
SAT_{FM_i} = (c_0 \land c_1 \land c_2 \land \ldots \land c_n) \land \text{dead feat}
\]

\[
SAT_{FM_{i+1}} = (c_0 \land \underline{c_1} \land c_2 \land \ldots \land c_n) \land \text{dead feat}
\]

\[
\vdots
\]

\[
SAT_{FM_{i+n}} = (\underline{c_0} \land c_1 \land c_2 \land \ldots \land c_n) \land \text{dead feat}
\]

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

\[
SAT_{FM_{\text{void}}} = (c_0 \land c_1 \land c_2 \land \ldots \land c_n)
\]

\[
SAT_{FM_{\text{core}}} = ((c_0 \land c_1 \land c_2 \land \ldots \land c_n) \land \neg \text{core feat})
\]

\[
\vdots
\]

\[
SAT_{FM_{\text{other}}} = ((\underline{c_0} \land c_1 \land c_2 \land \ldots \land c_n) \land \neg \text{other core feat})
\]

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

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

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

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

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

Our approach has many benefits: (1) End-users are only required to provide a single variational formula, which represents a set of related propositional formulas, rather than a formula and a hand-written program to direct the solver. (2) It is general; while variational satisfiability solving is applied to feature model analyses in this work, it can be used for any analysis that can be encoded as a variational formula. (3) With a variational formula, new kinds of syntactic manipulations, such as factoring out shared terms, become possible and can be automated. (4) A variational model may be produced that encapsulates a set of satisfying assignments for all variants of the variational formula, alleviating the need to track the incremental solver’s results when satisfying assignments 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 analyses. (Section 5.2)

We report a performance improvement over standard methods when solving many variants, and demonstrate variational void, core, and dead feature analyses. (Section 5.2)

2 BACKGROUND

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

After the discovery of the meltdown and spectre security vulnerabilities, there were multiple versions of the Linux kernel that dealt with these vulnerabilities (or not) in different ways. Suppose, for example, we have kernel versions \(L_0\), \(L_1\), and \(L_2\) with corresponding feature models \(FM_0\), \(FM_1\), and \(FM_2\). \(FM_0\) contains no spectre/meltdown-related formulas; \(FM_1\) contains a set of new formulas 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 \(FM_1\).

We introduce some notation to track particular features and propositional formulas across multiple feature models. For features we use \(f_{ij}\) to refer to the \(i\)th feature in the \(j\)th feature model. For formulas, we use \(\phi_{ij}\) to refer to the formula that encodes the \(i\)th feature’s relationships to other features in the \(j\)th feature model. When the feature model version is omitted, e.g., \(\phi_i\), we assume that \(\phi_i\) is unchanged and present in all feature models. Thus, the feature models can be represented by the following formulas:

\[
FM_0 = \phi_{0,0} \land \phi_{1,0} \land \ldots \land \phi_{n,0}
\]

\[
FM_1 = (\text{spectre_v2} \lor \text{l1tf}) \iff (\phi_{0,0} \land (\text{nospec_store_bypass_disable} \\
\implies f_{1j}) \land \phi_{1,0} \land (\text{pti} \implies \phi_{i,0}) \land \ldots \land \phi_{n,0})
\]

\[
FM_2 = \text{mitigations} \iff (\phi_{0,0} \land \phi_{1,0} \land \ldots \land \phi_{n,0})
\]

\(FM_1\) 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, \(\phi_{0,0}\) is constrained with a new conjunction, and there are three new formulas: (\(\phi_i \implies \phi_{i,0}\)), (\(\text{spectre_v2} \lor \text{l1tf}\)), (\(\text{nospec_store_bypass_disable} \implies f_{ij}\)), two of which affect a relationship or feature from \(FM_0\). In \(FM_2\), the features and constraints introduced in \(FM_1\) are replaced by a single new mitigations feature that is added to an unchanged copy of \(FM_0\).

Suppose one wants to find a satisfying assignment (i.e., a model) for each formula. If done with a classic SAT solver, then the procedure illustrated in Figure 1a results; where SAT solving is a batch process and no information is reused. Alternatively, a procedure using an incremental SAT solver is illustrated in Figure 1b; in this scenario, all of the formulas are solved by single solver instance where terms are programatically 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 SAT solver called an assertion stack. The assertion stack is a stack of declarations, definitions, or formulas that determine the context of the solver. A solver context is the union of all global variable

2 The feature names are from the actual Linux kernel, see [42].
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 ($\mathcal{C}_2$) with a choice construct from the choice calculus [29, 68], a formal language for describing variation. We call the new logic VPL, short for variational propositional logic, and refer to VPL expressions as variational formulas. This section defines the syntax and semantics of VPL and uses it to encode the example from Section 2.

Syntax. The syntax of variational propositional logic is given in Figure 2a. It extends the propositional formula notation of $\mathcal{C}_2$ with a single new connective called a choice from the choice calculus. A choice $D(f_1, f_2)$ 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 $\mathcal{C}_2$, which may be referenced in the leaves of a formula. We use lowercase letters to range over variables and uppercase letters for dimensions.

Figure 1

definitions and everything on the assertion stack. A program may add an assertion to the stack via the push operation and remove from the top via a pop operation [51].

In an efficient process one would initially add as many shared terms as possible, $\mathcal{F}M_0$ in this example. Then request a model, and manipulate the assertion stack to reach the next problem of interest, $\mathcal{F}M_1$ in this case. Notice that to reach the next problem, $\mathcal{F}M_1$, from $\mathcal{F}M_0$, several operations are required: $c_0, 0$ and $c_1, 0$ must be removed, $c_0, 0$ must be updated, and the new sub-formulas must be introduced. To reach $\mathcal{F}M_2$ from $\mathcal{F}M_1$ all assertions would need to be popped to add mitigation, then re-pushed.

(c) VPL equivalence laws

Figure 2: Formal definition of VPL.

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 $\mathcal{C}_2$, where choices correspond to a disciplined [43] application of #ifdef annotations. From a logical perspective, following the many-valued logic of Kleene [56], the intuition behind VPL is that a choice is semantically resolved by reference to an external environment. In this sense, VPL deviates from other many-valued logics, such as modal logic [33], because a choice waits until there is enough information to choose an alternative (i.e., until the formula is configured).

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

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

To illustrate the semantics of VPL, consider the formula \( p \land A(q,r) \), which has two variants: \( p \land q \) when \( A(C(A)) = \text{true} \) and \( p \land r \) when \( A(C(A)) = \text{false} \). From the semantics, it follows that choices in the same dimension are synchronized while choices in different dimensions are independent. For example, \( A(p,q) \land B(r,s) \) has four variants, while \( A(p,q) \land A(r,s) \) has only two \( (p \land r \land q \land s) \). 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, \( A(p,A(r,s)) \) is equivalent to \( A(p,s) \) 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 \( A(p \lor q, p \lor r) \) by first applying the \( \text{Os} \) law to obtain \( A(p,p) \lor A(q,r) \), then applying the \( \text{IPEMP} \) law to the first argument to obtain \( p \lor A(q,r) \) 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 since often we must apply an arbitrary number of laws right-to-left in order to set up a particular sequence of left-to-right applications that factor out commonalities.

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

\[
FM_{02} = L_2(\text{mitigations}, T) \equiv c_0 \land c_1 \land \ldots \land c_n
\]

We exploit the fact that \( \land \) forms a monoid with \( T \) to recover a formula equivalent to \( FM_0 \) for configurations where \( L_2 \) is false.

Next we combine \( FM_{02} \) 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 \( L_1 \):

\[
f_{FM_{012}} = L_1((\text{spectre_v2} \lor \text{ltfs}), L_2(\text{mitigations}, T))
\]

\[
\equiv L_1((c_{0,0} \land (\text{nospec_store_bypass-disable} \rightarrow f_1), c_{0,0})
\land L_1(c_{1,0}, T) \land c_1 \land L_1((\text{pft} \rightarrow c_{1,1}), T) \land \ldots \land c_n
\]

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

4 VARIATIONAL SATISFIABILITY SOLVING

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

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

We present an overview of a variational solver as a state diagram in Figure 3a that operates on the input’s abstract syntax tree. Labels on incoming edges denote inputs to a state and labels on outgoing edges denote return values; we show only inputs for recursive edges; labels separated by a comma share the edge. We omit labels that can be derived from the logical properties of connectives, such

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

\footnote{https://github.com/lambda-land/VSat-Papers/tree/master/SPLC2020}
The IL includes two kinds of terminals not present in the input query.
The variational solver translates the query formula to a formula that captures the variational structure of a query formula.

Derivation of a Variational Core. A variational core is an IL formula that captures the variational structure of a query formula. Plain terms will either be placed on the assertion stack or will be symbolically reduced, leaving only logical connectives, symbolic references, and choices.

Reduction engine shown in Figure 3b will produce a variational core that will assert \((a \land b)\) in the base solver, thus pushing it onto the assertion stack and create a symbolic reference for \((p \land \neg q)\). This is done in two states: evaluation, which communicates to the base solver to process plain terms, and accumulation which is called by evaluation to create symbolic references.

Generating the core begins with evaluation. Evaluation will match on the root node: \(\land\), of \(f\) and recur following the \(v_1 \land v_2\) edge, where \(v_1 = (a \land b) \land A(e_1, e_2)\) and \(v_2 = (p \land \neg q) \lor B(e_3, e_4)\). The recursion processes the left child first. Thus, evaluation will again match on \(\land\) of \(v_1\) creating another recursive call with \(v'_1 = (a \land b)\) and \(v'_2 = A(e_1, e_2)\). Finally, the base case is reached with a last recursive call where \(v''_1 = a\), and \(v''_2 = b\). At the base case both \(a\) and \(b\) are references, thus evaluation will send \(a\) to the base solver, following the \(r, s, t\) edge, which returns \(\bullet\) for the left child.

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

The right child, \(v_2 = (p \land \neg q) \lor B(e_3, e_4)\) requires accumulation. Evaluation will match on the root \(\lor\), and send \((p \land \neg q) \lor B(e_3, e_4)\) to accumulation via the \(v_1 \land v_2\) edge. Accumulation has two self-loops, one to create symbolic references (with labels \(r, s, \ldots\)), and one to recur to values. Accumulation matches the root \(\lor\) and recurs on the self-loop with edge \(v_1 \lor v_2\). \(v_1 = (p \land \neg q)\), and \(v_2 = B(e_3, e_4)\). Processing the left child first, accumulation will recur again with \(v'_1 = p\) and \(v'_2' = \neg q\). \(v'_1\) is a base case for choices and cannot be reduced in evaluation, and so \(\bullet \land A(e_1, e_2)\) will be reduced to just \(A(e_1, e_2)\) as the result for \(v_1\).

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

The right child, \(v_2 = (p \land \neg q) \lor B(e_3, e_4)\) requires accumulation. Evaluation will match on the root \(\lor\), and send \((p \land \neg q) \lor B(e_3, e_4)\) to accumulation via the \(v_1 \land v_2\) edge. Accumulation has two self-loops, one to create symbolic references (with labels \(r, s, \ldots\)), and one to recur to values. Accumulation matches the root \(\lor\) and recurs on the self-loop with edge \(v_1 \lor v_2\). \(v_1 = (p \land \neg q)\), and \(v_2 = B(e_3, e_4)\). Processing the left child first, accumulation will recur again with \(v'_1 = p\) and \(v'_2' = \neg q\). \(v'_1\) is a base case for choices and cannot be reduced in evaluation, and so \(\bullet \land A(e_1, e_2)\) will be reduced to just \(A(e_1, e_2)\) as the result for \(v_1\).

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

The right child, \(v_2 = (p \land \neg q) \lor B(e_3, e_4)\) requires accumulation. Evaluation will match on the root \(\lor\), and send \((p \land \neg q) \lor B(e_3, e_4)\) to accumulation via the \(v_1 \land v_2\) edge. Accumulation has two self-loops, one to create symbolic references (with labels \(r, s, \ldots\)), and one to recur to values. Accumulation matches the root \(\lor\) and recurs on the self-loop with edge \(v_1 \lor v_2\). \(v_1 = (p \land \neg q)\), and \(v_2 = B(e_3, e_4)\). Processing the left child first, accumulation will recur again with \(v'_1 = p\) and \(v'_2' = \neg q\). \(v'_1\) is a base case for choices and cannot be reduced in evaluation, and so \(\bullet \land A(e_1, e_2)\) will be reduced to just \(A(e_1, e_2)\) as the result for \(v_1\).

In evaluation, conjunctions can be split because of the behavior of the assertion stack and the and-elimination property of \(\land\). Disjunctions and negations cannot be split in this way because both cannot be performed if a child node has been lost to the solver, e.g., \(\neg \bullet\). Thus, in accumulation, we construct symbolic terms to represent entire sub-trees, ensuring information is not lost, but still allowing for the sub-tree to be evaluated if it is sound to do so.
Symbolic references are variables in the reduction engine’s memory that represent a set of statements in the base solver. For example, \(s_{pq}\) represents three declarations in the base solver:

\[
\begin{align*}
(\text{declare-const } p \text{ Bool}) & \quad : \quad s_{pq} \text{ represents} \\
(\text{declare-const } q \text{ Bool}) & \quad : \quad \text{ several declarations} \\
(\text{declare-fun } s_{ab} (\text{Bool} \text{ or } p \text{ not } c)) & : \quad \text{ fill}
\end{align*}
\]

Similarly a variational core is a sequence of statements in the base solver with holes \(\Diamond\). For example, the representation of \(\text{VCore}_j\):

\[
\begin{align*}
(\text{assert } (\text{and } a \text{ b})) & \quad : \quad a \land b \text{ on the assertion stack} \\
(\text{assert } \Diamond) & : \quad \text{ choice } A \\
& \quad : \quad \text{ many declares may occur} \\
(\text{assert } \Diamond) & : \quad \text{ many assertions may occur} \\
(\text{declare-fun } s_{pq} () \text{ Bool } (\text{and } p \text{ q})) & \quad : \quad \text{ choice } B \\
(\text{assert } (\text{or } s_{ab} \Diamond)) & : \quad \text{ assert waiting on } [B(e_3, e_4)]_C.
\end{align*}
\]

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

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

We display a subset of edges of the reification engine using the \(\land\) connective. In general, these edges will be duplicated for each binary logical connective, e.g., \(\lor\). The left edge, is taken when a choice is observed in the variational core: \(a \land (\{D(e_3, e_4)\}_C \land D \in C)\). This edge reduces choices with dimension \(D\) to an alternative, which are then translated to IL. The right edge is dashed to indicate assertion stack manipulation, and is taken when \(D \notin C\). For this edge, the configuration is mutated for both alternatives: \(C \cup \{(D, T)\}\), and \(C \cup \{(D, F)\}\), and the recursive call is wrapped with a push, and pop command. To the base solver, this branching is a linear sequence of assertion stack manipulations that performs backtracking behavior, for example the representation of \(f\) is:

\[
\begin{align*}
\text{push 1} & : \quad \text{ declares and assertions from VCore} \\
\text{pop 1} & : \quad \text{ a configuration on B has occurred} \\
\text{push 1} & : \quad \text{ new declarations for left alternative} \\
\text{pop 1} & : \quad \text{ return for right alternative} \\
\text{push 1} & : \quad \text{ repeat for right alternative}
\end{align*}
\]

\(f_0 \rightarrow T\)
\(f_1 \rightarrow F\)
\(f_n \rightarrow F\)
\(C_{FT} = \{(L_1, F), (L_2, T)\}\)

Figure 4: Possible plain models for variants of \(f_{\text{FM}}\),

\[
\begin{align*}
_Sat & \rightarrow (L_1 \land L_2) \lor (\neg L_1 \land \neg L_2) \\
f_0 & \rightarrow (L_1 \land L_2) \lor (\neg L_1 \land \neg L_2) \\
f_1 & \rightarrow (\neg L_1 \land \neg L_2)
\end{align*}
\]

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

Where the hole \(\Diamond\), will be filled with a newly defined variable \(s_{DF}\) that represents the left alternative’s formula.

**Variational Models.** Plain models map variables to Boolean values; variational models map variables to variation contexts that record the variants where the variable was assigned \(T\). We denote the variation context for a variable \(r\) as \(\text{vc}_r\), and maintain a variable called \(\_Sat\) to track which configurations are satisfiable. As an example, consider the query formula \((f_{\text{FM}})\) from the Linux example in Section 2. If each variant is satisfiable, there are three models, as illustrated in Figure 4; the corresponding variational model is shown in Figure 5. \(\text{vc}_{\_Sat}\) consists of three disjuncted terms, one for each satisfiable variant. A satisfiable assignment of the query formula can be found by calling \(\text{SAT}()\) on \(\text{vc}_{\_Sat}\). Assuming the model \(C_{FT} = \{(L_1, F), (L_2, T)\}\) is returned, substitution on \(\text{vc}_{f_1}\) yields \(f_1\)’s value in \(C_{FT}\):

\[
\begin{align*}
f_1 & \rightarrow (\neg L_1 \land \neg L_2) \\
\text{ve for } f_1 & \\
\text{ve for } f_1 & \rightarrow (\neg F \land T) \\
\text{ve for } f_1 & \rightarrow (L_1 \land T) \\
\text{ve for } f_1 & \rightarrow (L_2 \land T)
\end{align*}
\]

Furthermore, finding variants where a variable \(f_j\) is satisfiable reduces to \(\text{SAT}(\text{vc}_{f_j})\).

Variational models are constructed incrementally by merging each new plain model returned by the solver into the variational model. A merge requires the current configuration, the plain model, and current \(\text{vc}\) of a variable. Variables are initialized to \(F\). For each variable \(i\) in the model, if \(i\)’s assignment is \(T\) in the plain model,
then the configuration is translated to a variation context and disjunctioned with \( v_c \). For example, to merge the \( CRF' \)'s plain model to the variational model in Figure 5, \( CRF' \)'s configuration is converted to \( \neg L_1 \land L_2 \). This clause is disjuncted for variables assigned \( T \) in the plain model: \( v_c_0, \neg v_c_1, \) and \( v_c_mitigations \), even if they are new (e.g., mitigations). Variables assigned \( F \) are skipped, thus \( v_c_2 \) remains \( F \).

In the next model \( CRF''_\_1 \_q \) thus \( v_c_2 \) remains unaltered. Variables such as \( f_\_t \), whose \( v_c \)'s stay \( F \) are called constant.

5 QUANTITATIVE EVALUATION

Section 4 provides a technique for variational solving that enables sharing work on subterms that are common across several variants. However, the technique also involves substantial overhead, so it is not obvious that it leads to performance gains in realistic problems. To investigate, we construct a prototype variational solver, VSAT in the Haskell programming language [35] and quantitatively compare it to incremental and non-incremental SAT solving. We reuse real-world data from a previous study by Nieke et al. [53]. Nieke et al.'s study provides two datasets, automotive02 and financialServices1, which encode the evolution histories of two feature models as propositional formulas.\(^3\) We refer to these as the auto dataset, and fin dataset for the remainder of the paper.

5.1 Experimental Methodology

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

For example, and to demonstrate the flexibility of variational solving, we construct a VPL formula that encodes both a dead analysis and core analysis over all features \( f \) in a query formula \( q \) by introducing a choice with a new dimension \( DC \) that does not correspond to any version: \( q_{DC} = q \land DC(\land \_f eq f_\_1 \land \_f eq \neg f_\_1) \). If \( q \) encodes several variants identified by dimensions \( V_0, \ldots, V_n \), then \( q_{DC} \) contains dimensions that correspond to two different concepts in the application domain \( V_i \) for versions and \( DC \) for the kind of analysis. Selecting an analysis is then performed by a \( v_c \):\(^6\) exactly \( V_0, \ldots, V_n \). The \( v_c \) selects exactly one version variant with \( exactly_{V_0, \ldots, V_n} \) but leaves the dimension \( DC \) undefined. With \( DC \) undefined, VSAT will try both \( DC \) set to \( T \) and \( DC \) set to \( F \). Thus, the \( v_c \) selects exactly two variants per version variant, one for the core analysis and one for the dead analysis. To include a void analysis, in addition to the core and dead analyses, another choice is required: \( q_{VDC} = q \land Void(\land \_f eq f_\_1 \land \_f eq \neg f_\_1, T) \).

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

**RQ1** How does variational solving scale as variation increases?

**RQ2** What is the impact of sharing on performance?\(^7\)

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

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

Data Description and Encoding. Nieke et al.'s formulas collapse sets of \( C_2 \) formulas to a single formula using implications on an SMT variable that represents a moment in time. A two-pass process was used to translate Nieke et al.'s formulas into VPL—one pass to parse to an internal representation and another to detect and convert Nieke et al.'s temporal ranges to choices, nesting the implied clauses into the true alternative. The two-pass process conserves Nieke et al.'s ordering of plain terms and encoding. The two datasets differ in important ways. The auto dataset encodes four monthly snapshots while the fin dataset encodes ten. Hence, the auto's query formula represents 16 variants, while the fin query formula represents 1,024 variants. For \( RQ2 \) and \( RQ3 \), we construct several \( v_c \)'s to restrict the analysis to version variants. The \( v_c \)s range from ones that enable only one version variant (for \( RQ3 \)): \( fm_{auto} V_1 = (V_1 \land \neg V_2 \land \neg V_3 \land \neg V_4) \) to \( V_c \)s that enable only consecutive version variants (for \( RQ2 \)): \( fm_{auto} V_{12} = V_1 \lor V_2 \).

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

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

We construct the \( p \rightarrow p \) algorithm by configuring the query formula to its version variants before benchmarking begins. These formulas are then sent to the base solver one-by-one, without the solver maintaining information between queries. To assess the potential overhead of solving a plain query on a variational solver, the \( p \rightarrow v \) case and corresponding to \( RQ3 \), we perform the same pre-processing as the \( p \rightarrow p \) case but send each plain formula to VSAT instead. This provides insight into the cost incurred by the reduction engine. For \( v \rightarrow p \), we configure the query formula to
retrieve version variants during benchmarking. Each formula is sent to the base solver with the solver maintaining information between queries. This gives insight into the overhead incurred by configuring a variational formula and the benefits of caching.

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

Unless specified, all results are a bootstrapped statistical average.7 For RQ2, we normalize the data to the baseline (v→p), fit a linear model, and statistically assess differences of samples by performing a one-way Kruskall-Wallis test [54] followed by a pairwise Wilcoxon test [70] with Bonferroni p-value correction [27]. For RQ3, we retrieve the 10 raw measurements from the bootstrapped versions. Figure 6 shows the ratio of unsatisfiable models to total terms for each [version] (as represented by variant count). For both datasets the number of satisfiable models decreased as new [versions] were considered, and the majority of features remained constant. Thus, the variational model is likely a compressed version of the set of plain models. Compression metrics were not calculated as this is an orthogonal concern to the performance of variational satisfiability solving.

Variational models permit product analyses without a SAT solver. Figure 6 shows such a purely syntactic analysis: counting disjuncted clauses in the variational model as a representation of satisfiable plain models. We believe post-hoc analyses such as this may be useful to feature modelers as they direct attention to impactful versions of the feature model. For example, the change to V8 from

5.2 Results and Discussion

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

Variational models permit product analyses without a SAT solver. Figure 6 shows such a purely syntactic analysis: counting disjuncted clauses in the variational model as a representation of satisfiable plain models. We believe post-hoc analyses such as this may be useful to feature modelers as they direct attention to impactful versions of the feature model. For example, the change to V8 from 1

67 \times 10^{-4},\text{and }1.10 \times 10^{-4}\text{ thus confirming that sharing positively correlates to speedups for variational solving in these datasets.}

This result is further evidence that as the reduction engine reduces more of the query formula, more reuse occurs, such as observed in the auto dataset where the sharing ratio is high. Hence, an avenue of future work is to leverage the laws of the variational logic to automatically refactor input formulas. The consequences of this observation will be particular to the application domain.

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

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

V7 (128 to 256 Variants) of fin clearly constrained the feature model, decreasing the number of constant features.

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

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

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

This result is further evidence that as the reduction engine reduces more of the query formula, more reuse occurs, such as observed in the auto dataset where the sharing ratio is high. Hence, an avenue of future work is to leverage the laws of the variational logic to automatically refactor input formulas. The consequences of this observation will be particular to the application domain.

1Using v0.2.5 of the gauge [55] library and v5.7.1 of the z3 [26] SMT solver with a solver seed set to 1728. All data was collected on a server running CentOS Linux release 7.7, with two Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz, 512GB RAM. We used stack lts-15.7 (GHC 8.8.3) and tested with RTS options “-gg -A64m -AL128m -n8m”.

We omit the matrix of pairwise comparisons from the paper for auto VSAT did not exhibit significant overhead for the plain formulas but ensured each algorithm experienced identical ordering of plain terms as described in Section 5.1.

Our work is most similar to Similar Solvers, Related Techniques. Our work is most similar to Visser et al. [66], which also constructs a SAT solver that exploits encoding-based threats to validity by reusing Nieke et al.’s formulas but ensured each algorithm experienced identical ordering of plain terms as described in Section 5.1.

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

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

In this paper, we do not provide a proof of the soundness of our methods. We mitigate this threat in several ways: we performed property-based testing [22] on our prototype and verified that a satisfiable variant was found to be satisfiable across all algorithms. In addition, we define a property that ensures that for each plain formula, found with p→∅, we can obtain a core, dead, and void analysis without a core. Notably, we are unable to make absolute performance claims because our study, with only two product lines, may not be representative.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>v→v</th>
<th>v→p</th>
<th>p→v</th>
<th>p→p</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>211.70</td>
<td>288.66</td>
<td>363.16</td>
<td>378.69</td>
</tr>
<tr>
<td>fin</td>
<td>11.1</td>
<td>8.42</td>
<td>8.07</td>
<td>9.51</td>
</tr>
</tbody>
</table>

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

Variational Dead and Core Analysis. Table 1 displays the performance results for the dead and core analyses. We observe a 76% speedup for the auto dataset, and a 36% slowdown for fin dataset. This difference is due to the threshold at which VSAT begins to outperform other algorithms. For auto this threshold was low, at 4 variants, but was 64 variants for the fin dataset, thus the slowdown. Following RQ2’s results, had a core, dead, and void analysis been performed, v→v would still be under the speedup threshold.

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

For software product lines this means that any method to increase sharing between versions or SAT problems is desirable; this may be smaller changes with respect to the entire feature model, more frequent snapshots of the feature model, or syntactic manipulations to mitigate the occurrence of new features.

This overhead is particular to some formulas, suggesting certain formula characteristics may have a large effect. Identifying these characteristics requires a more robust dataset; that some variants show no overhead suggests future work to recover performance.

For software product lines this means that any method to increase sharing between versions or SAT problems is desirable; this may be smaller changes with respect to the entire feature model, more frequent snapshots of the feature model, or syntactic manipulations to mitigate the occurrence of new features.

**RQ3: Overhead of a Plain Query on VSAT.** Figure 9 displays the bootstrapped averages of each version variant for each algorithm. We omit the matrix of pairwise comparisons from the paper for space, although it is available online. Of a total of 84 comparisons, 23 were significant in fin and 2 in auto. Given RQ2, and the composition of fin, we expect VSAT to show slowdowns for fin. This is observed in Figure 9 and is statistically significant for all versions. For auto, the only differences were in V1, and between (p→v, v→v) and (p→v, v→p). Notably, v→v did not differ from v→p, thus VSAT did not exhibit significant overhead for the auto dataset. That p→v was statistically different for V1 suggests particular formulas may not respond well to the reduction engine. Similarly, there is clearly overhead when solving plain formulas, although this overhead is particular to some formulas, suggesting certain formula characteristics may have a large effect. Identifying these characteristics requires a more robust dataset; that some variants show no overhead suggests future work to recover performance.

![Figure 8: Sharing positively correlates to speedup only for v→v, where % SpeedUp = Algorithm v→p.](image1)

![Figure 9: v→v incurs an average slowdown of 17% for auto, and 60% for the fin, when solving a single version.](image2)
which have been successfully applied to information-flow security. The variability of SPLs or configurable software is often reduced by analyzing versions of a feature model in a single formula. We reuse their learned clauses in the solver, it loses the ability to increase reuse of learned clauses in the solver, it loses the ability to reuse SAT solving. Since the size of SAT problems in software variability applications is often dominated by the feature model, researchers tried to reduce the size of satisfiability problems by delaying consideration of the feature model until after the analysis and only using it if false positives are found. This technique increases the overall efficiency of static analysis. While Classen et al. [23] found that it actually decreases efficiency of family-based model checking. Variational solving is orthogonal to these approaches since the feature model can be excluded from a variational formula and then used later to rule out false positives. Feature models can also be reduced in size to speed up analyses, for example, by slicing or decomposition. 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 implication graphs, which support faster reasoning. The idea is to encode as many software variability constraints as possible in such graphs, then use a SAT solver only for the remaining constraints. The construction of modal implication graphs already requires solving SAT problems, but this cost is amortized if many SAT queries will be solved during the analysis, as Krieter et al. [40] found for configuration processes.

7 CONCLUSION
Variational satisfiability solving offers numerous advantages over current methods. Variational models, as solutions to variational satisfiability problems, are a flexible, compressed representation that enables post-hoc analyses. Through the use of a VPL formula, variational solving provides a domain agnostic, automated approach to use an incremental solver to efficiently solve sets of SAT problems, in addition to making explicit the ordering between plain and variational terms. Furthermore, we have demonstrated that sharing can achieve the reuse benefits of the global approach without sacrificing the precision of the local approach.

While the magnitude of its effect is not yet known, our analysis forms a foundation for future research. For feature modelers, variational satisfiability solving offers the practical benefits of a faster and more flexible analysis tool, and provides a basis for new kinds of automated variational analyses on feature models and software product lines. Outside the domain of software product lines, variational satisfiability solving provides a framework and logic where variation can be directly represented irrespective of the application domain, thus providing a new method to study variation itself.

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


