AN ABSTRACT OF THE DISSERTATION OF

Jeffrey M. Young for the degree of Doctor of Philosophy in Computer Science
presented on September 23, 2011.

Title: Variational Satisfiability Solving

Abstract approved: _____

Eric Walkingshaw

Over the last two decades, satisfiability and satisfiability-modulo theory (SAT/SMT) solvers have grown powerful enough to be general purpose reasoning engines throughout software engineering and computer science. However, most practical use cases of SAT/SMT solvers require not just solving a single SAT/SMT problem, but solving sets of related SAT/SMT problems. This discrepancy was directly addressed by the SAT/SMT community with the invention of incremental SAT/SMT solving. However, incremental SAT/SMT solvers require end-users to hand write a program which dictates the terms that are shared between problems and terms which are unique. By placing the onus on end-users, incremental solvers couple the end-users' solution to the end-users' *exact* sequence of SAT/SMT problems—making the solution overly specific—and require the end-user to write extra infrastructure to coordinate or handle the results.

This dissertation argues that the aforementioned problems are caused from the lack of variation as a computation concept, similar to that of a WHILE loop. To demonstrate

the argument, this thesis applies theory from *variational* programming to the domain of SAT/SMT solvers to create the first variational SAT solver. The thesis formalizes a variational propositional logic and specifies variational SAT solving as a transpiler, which transpiles variational SAT problems to non-variational SAT that are then processed by an industrial SAT solver. It shows that the transpiler is an instance of a variational fold and uses that fact to extend the variational SAT solver to an asynchronous variational SMT solver. Finally, it defines a general algorithm to construct a single variational string from a set of non-variational strings.

Variational Satisfiability Solving

by

Jeffrey M. Young

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented September 23, 2011
Commencement June 2012

Doctor of Philosophy dissertation of Jeffrey M. Young presented on September 23, 2011.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Jeffrey M. Young, Author

ACKNOWLEDGEMENTS

I would like to acknowledge the Starting State and the Transition Function.

TABLE OF CONTENTS

LIST OF FIGURES

Todo list

Chapter 1: Introduction

One of the most important aspects of any programming language is the ability to control complexity, especially as software written in that language grows. The burgeoning field of *variation theory* and *variational programming* [7, 12, 13, 18, 36] attempt to control complexity which is induced into a software artifact when many *similar yet distinct* kinds of the same software artifact must coexist. For example, software is often *ported* to other platforms, creating similar, yet distinct instances of that software which must be maintained. Such instances of variation are ubiquitous: Web applications are tested on multiple servers; programming languages maintain backwards compatibility and so do software libraries; databases evolve over time, locale and data; and device drivers must work with varying processors and architectures. Variation theory and variational programming have been successful in small systems, yet it has not been tested in a performance demanding practical domain. In the words of Joe Armstrong [2], "No theory is complete without proof that the ideas work in practice"; this is the central project of this thesis, to put the ideas of *variation* and *variational programming* to the test in the practical domain of satisfiability solving (SAT).

cite variation
data structur
and images

The major contribution of this thesis is the formalization of a *VPL*, *variational satisfiability solving*, and the construction of a *variational SAT solver*. In the next section I motivate the use of variation theory and variational techniques in satisfiability

solving. In addition to work on variational SAT several other contributions are made. The thesis extends variational satisfiability solving to variational satisfiability-modulo theories (SMT). It demonstrates reusable techniques and architecture for constructing *variational or variation-aware* systems using the non-variational counterparts of these systems for other domains. It shows that, with the concept of variation, the variational SMT and SAT solvers can be trivially parallelized. Lastly, the thesis provides a general algorithm to construct variational strings from a set of non-variational strings and argues for the proliferation of variation theory to other domains in computer science.

1.1 Motivation and Impact

Classic SAT, which solves the boolean satisfiability problem [5] has been one of the largest success stories in computer science over the last two decades. Although SAT solving is known to be NP-complete [8], SAT solvers based on conflict-driven clause learning (CDCL) [3, 23, 29] have been able to solve boolean formulae with millions variables quickly enough for use in real-world applications [33]. Leading to their proliferation into several fields of scientific inquiry ranging from software engineering to Bioinformatics [16, 22].

The majority of research in the SAT community focuses on solving a single SAT problem as fast as possible, yet many practical applications of SAT solvers [6, 9, 10, 14, 28, 30, 37] require solving a set of related SAT problems [10, 28, 30]. To take just one example, software product-lines (SPL) utilizes SAT solvers for a diverse range of analyses

including: automated feature model analysis [4,15,32], feature model sampling [25,34], anomaly detection [1, 20, 24], and dead code analysis [31].

This misalignment between the SAT research community and the practical use cases of SAT solvers is well known. To address the misalignment, modern solvers attempt to propagate information from one solving instance, on one problem, to future instances in the problem set. Initial attempts focused on clause sharing (CS) [28, 37] where learned clauses from one problem in the problem set are propagated forward to future problems. Although, modern solvers are based on a major breakthrough that occurred with *incremental SAT under assumptions*, introduced in Minisat [11].

Incremental SAT under assumptions, made two major contributions: a performance contribution, where information including learned clauses, restart and clause-detection heuristics are carried forward. A usability contribution; Minisat exposed an interface that allowed the end-user to directly program the solver. Through the interface the user can add or remove clauses and dictate which clauses or variables are shared and which are unique to the problem set, thus directly addressing the practical use case of SAT solvers.

Despite the its success, the incremental interface introduced a programming language that required an extra input, the set of SAT problems, *and* a program to direct the solver with side-effectual statements. This places further burden on the end-user: the system is less-declarative as the user must be concerned with the internals of the solver. A new class of errors is possible as the input program could misuse the introduced side-effectual statements. By requiring the user to direct the solver, the users' solution is specific to the exact set of satisfiability problems at hand, thus the programmed solution

is specific to the problem set and therefore to the solver input. Should the user be interested in the assignment of variables under which the problem at hand was found to be satisfiable, then the user must create additional infrastructure to track results; which again couples to the input and is therefore difficult to reuse.

I argue that solving a set of related SAT problems *is a variational programming problem* and that by directly addressing the problem's variational nature the incremental SAT interface and performance can be improved. The essence of variational programming is a formal language called the *choice calculus*. With the choice calculus, sets of problems in the SAT domain can be expressed syntactically as a single *variational artifact*. The benefits are numerous:

1. The side-effectual statements are hidden from the user, recovering the declarative nature of non-incremental SAT solving.

2. Malformed programs built around the control flow operators become syntactically impossible.

3. The end-user's programmed solution is decoupled from the specific problem set, increasing software reuse.

4. The solver has enough syntactic information to produce results which previously required extra infrastructure constructed by the end-user.

5. Previously difficult optimizations can be syntactically detected and applied before the runtime of the solver.

This work is applied programming language theory in the domain of satisfiability solvers. Due to the ubiquity of satisfiability solvers estimating the impact is difficult although the surface area of possible applications is large. For example, many analyses in the software product-lines community use incremental SAT solvers. By creating a variational SAT solver such analyses directly benefit from this work, and thus advance the state of the art. For researchers in the incremental satisfiability solving community, this work serves as an avenue to construct new incremental SAT solvers which efficiently solve classes of problems that deal with variation.

For researchers studying variation the significance and impact is several fold. By utilizing results in variational research, this work adds validity to variational theory and serves as an empirical case study. At the time of this writing, and to my knowledge, this work is the first to directly use results in the variational research community to parallelize a variation unaware tool. Thus by directly handling variation, this work demonstrates direct benefits to be gained for researchers in other domains and magnifies the impact of any results produced by the variational research community. Lastly, the result of my thesis, a variational SAT solver, provides a new logic and tool to reason about variation itself.

1.2 Contributions and Outline of this Thesis

The high-level goal of this thesis is to use variation theory to formalize and construct a variational satisfiability solver that understands and can solve SAT problems that con-

tain *variational values* in addition to boolean values. It is our desire that the work not only be of theoretical interest but of practical use. Thus, the thesis provides numerous examples of variational SAT and variational SMT problems to motivate and demonstrate the solver. The rest of this section outlines the thesis and expands on the contributions of each chapter:

1. Chapter 2 (*Background*) provides the necessary material for a reader to understand the contributions of the thesis. This section provides an overview of satisfiability solving, satisfiability-modulo theories solving, incremental SAT and SMT solving . Several important concepts are introduced: The definition of satisfiability and definition of the boolean satisfiability problem. The internal data structure incremental SAT solvers utilize to provide incrementality, and the side-effectual operations which manipulate the incremental solver and form the basis of variational satisfiability solving. Lastly, the definition of the output of a SAT or SMT solver which has implications for variational satisfiability solving and variational SMT.

2. Chapter 3 (*Variational Propositional Logic*) introduces a variational logic that a variational SAT solver operates upon. This section makes several important contributions:

   (a) Introduces the essential aspects of variation using propositional logic and in the process presents the first instance of a recipe to construct a *variation-aware* version of a system using the non-variational version.

(b) Defines and formalizes variational concepts that are used throughout the thesis, such as *variant*, *configuration* and *variational artifact*.

(c) Proves theorems that are central to the soundness of variational satisfiability solving, such as the property of *variation-preservation*

3. Chapter 4 (*Variational Satisfiability Solving*)

Chapter 2: Background

Chapter 3: Variational 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 [12, 35], 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 **??**.

Syntax The syntax of variational propositional logic is given in Fig. 3.1a. 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.

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

(a) Syntax of VPL.

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

(b) Configuration semantics of VPL.

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

(c) VPL equivalence laws.

Figure 3.1: Formal definition of VPL.

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 [21] application of `#ifdef` annotations. From a logical perspective, following the many-valued logic of Kleene [19, 27], the intuition behind VPL is that a choice is a placeholder for two equally possible alternatives that is deterministically resolved by reference to an external environment. In this sense, VPL deviates from other many-valued logics, such as modal logic [17], 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 Fig. 3.1b and describes how choices are eliminated from a formula. The semantics is parameterized by a *configuration* C, which is a partial function from dimensions to Boolean values. The first four cases of the semantics simply propagate configuration down the formula, terminating at the leaves. The case for choices is the interesting one: if the dimension of the choice is defined in the configuration, then the choice is replaced by its left or right alternative corresponding to the associated value of the dimension in the configuration. If the dimension is undefined in the configuration, then the choice is left intact and configuration propagates into the choice's alternatives.

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

possible total configuration) the *variants* of f.

To illustrate the semantics of VPL, consider the formula $p \wedge A\langle q, r\rangle$, which has two variants: $p \wedge q$ when $C(A) = \mathsf{true}$ 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. Fig. 3.1c defines a set of equivalence laws for VPL formulas. These laws follow directly from the configuration semantics in Fig. 3.1b 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 Fig. 3.1c. 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[1] 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

[1] We hypothesize that it is equivalent to BDD minimization, which is NP-complete, but the equivalence has not been proved; see [36].

true. This yields the following variational formula.

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

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

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

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

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 **??**.

Chapter 4: Variational Satisfiability Solving

In this section, we present our algorithm for variational satisfiability solving. Sec. 4.0.1 provides an overview of the algorithm and introduces the notion of *variational models* as solutions to variational satisfiability problems. **??** provides the formal specification.

4.0.1 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 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 time the entire term is consumed 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. The models for each variant are combined into a single *variational model* that captures the result of solving all variants of the original VPL formula.

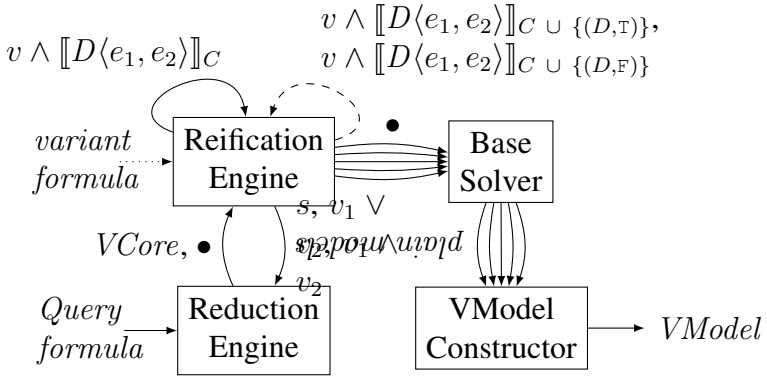

Figure 4.1: System overview of the variational solver.

We present an overview of the variational solver as a state diagram in Fig. 4.1 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 as commutativity of $\vee$ and $\wedge$. Similarly, we omit base case edge labels for choices and describe these cases in the text.

The solver has four subsystems: The *reduction engine* processes plain terms and generates the variational core, which is ready for reification. The *reification engine* configures choices in a variational core. The *base solver* is the incremental solver used to produce plain models. Finally, the *variational model constructor* synthesizes a single variational model from the set of plain models returned by the base solver.

The solver takes a VPL formula called a *query formula* and an optional input called a *variation context* (vc). A vc is a propositional formula of dimensions that restricts

the solver to a subset of variants. The variational solver translates the query formula to a formula in an intermediate language (IL) that the reduction and reification engines operate over. The syntax of the IL is given below.

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

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

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

The variational core for a VPL formula is computed by a reduction engine illustrated in Fig. 4.2. 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 $f = ((a \wedge b) \wedge A\langle e_1, e_2\rangle) \wedge ((p \wedge \neg q) \vee B\langle e_3, e_4\rangle)$. Translated to an IL formula, f has four references (a, b, p, q) and two choices. The reduction engine will ultimately produce a variational core that asserts $(a \wedge b)$ in the base solver, thus pushing it onto the assertion stack, and create a symbolic reference for $(p \wedge \neg q)$.

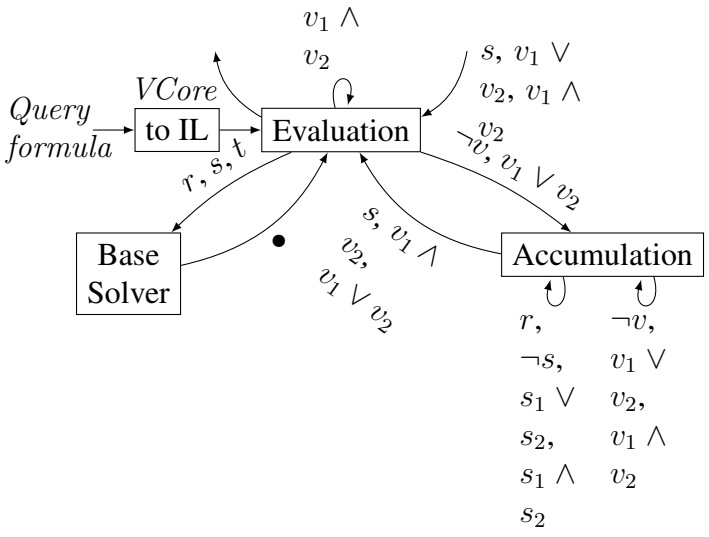

Figure 4.2: Overview of the reduction engine.

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

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

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

A variational core is derived to save redundant work. If solved naively, plain sub-

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

Symbolic references are variables in the reduction engine's memory that represent a set of statements in the base solver.[1] For example, 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)))
```

Similarly a variational core is a sequence of statements in the base solver with holes $\Diamond$. For example, the variational core of f would be encoded as:

```
(assert (and a b))                ;; add a ∧ b to the assertion stack
(declare-const ◇)                 ;; choice A
  ⋮                               ;; potentially many declarations and assertions
(declare-fun s_pq () Bool (and p q))  ;; get symbolic reference for s_pq
(declare-const ◇)                 ;; choice B
  ⋮                               ;; potentially many declarations and assertions
(assert (or s_ab ◇))              ;; assert waiting on ⟦B⟨e3, e4⟩⟧C
```

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

Solving the Variational Core The reduction engine performs the work at each recursive step whereas the reification engine defines transitions between the recursive steps by manipulating the configuration. In **??**, we formalized a configuration as a function $D \rightarrow \mathbb{B}$, which we encode in the solver as a set of tuples $\{D \times \mathbb{B}\}$. Fig. 4.1 shows

[1] 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 that exposes an incremental API as defined by minisat [26] can be used to implement variational satisfiability solving.

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 retrieved from the base solver when the reduction engine returns $\bullet$, indicating that a variant has been reached.

We show the edges of the reification engine relating to the $\wedge$ connective; the edges for the $\vee$ connective are similar. The left edge is taken when a choice is observed in the variational core: $v \wedge [\![D\langle e_1, e_2\rangle]\!]_C$ and $D \in C$. This edge reduces choices with dimension D to an alternative, which is 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: $C \cup \{(D, \texttt{T})\}$ and $C \cup \{(D, \texttt{F})\}$, and the recursive call is wrapped with a `push` and `pop` command. To the base solver, this branching appears as a linear sequence of assertion stack manipulations that performs backtracking behavior. For example, the representation of f is:

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

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

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

Figure 4.3: Possible plain models for variants of $f_{FM_{02}}$.

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

Figure 4.4: Variational model of the plain models in Fig. 4.3.

Variational Models Plain models map variables to Boolean values; variational models map variables to variation contexts that record the variants where the variable was assigned T. We denote the variation context for a variable r as vc_r, and maintain a special variable called $\_Sat$ to track which configurations are satisfiable.

For example, consider the query formula $f_{FM_{012}}$ from the Linux example in **??**. If each variant is satisfiable, there are three models, as illustrated in Fig. 4.3. The corresponding variational model is shown in Fig. 4.4. The variation context for $\_Sat$, $vc_{\_Sat}$, consists of three disjuncted terms, one for each satisfiable variant. A satisfiable assignment of the query formula can be found by calling SAT on $vc_{\_Sat}$. Assuming the model

$C_{FT} = \{(L_1, \texttt{F}), (L_2, \texttt{T})\}$ is returned, substitution on vc_{f_i} yields f_i's value in C_{FT}:

$f_i \rightarrow (\neg L_1 \wedge L_2)$	vc for f_i
$f_i \rightarrow (\neg \texttt{F} \wedge \texttt{T})$	Substitute F for L_1, T for L_2
$f_i \rightarrow \texttt{T}$	Result

Additionally, we can compute all variants where a variable f_j is satisfiable by solving $SAT(vc_{f_j})$

Variational models are constructed incrementally by merging each new plain model returned by the solver into the variational model. A merge requires the current configuration, the plain model, and the current vc of a variable. Variables are initialized to F. For each variable i in the model, if i's assignment is T in the plain model, then the configuration is translated to a variation context and disjuncted with vc_i. For example, to merge the C_{FT}'s plain model to the variational model in Fig. 4.4, C_{FT}'s configuration is converted to $\neg L_1 \wedge L_2$. This clause is disjuncted with the current vc for in the variational model for all of the variables assigned T in the plain model: vc_0, vc_i, and $vc_{mitigations}$, even if they are new (e.g., $mitigations$). Variables assigned F are skipped, thus vc_n remains F. In the next model C_{TT}, f_i is F so vc_i remains unaltered. Variables such as f_n, whose vc remains F, are called *constant*.

Chapter 5: Variational Satisfiability-Modulo Theory Solving

Chapter 6: Case Studies

Chapter 7: Related Work

Chapter 8: Conclusion

Bibliography

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

[2] Joe Armstrong. Making reliable distributed system in the presence of software errors. Website, 2003. Available online at http://www.erlang.org/download/armstrong_thesis_2003.pdf; visited on March 16th, 2021.

[3] Roberto J. Bayardo and Robert C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 203–208. AAAI Press, 1997.

[4] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated Reasoning on Feature Models. pages 491–503, 2005.

[5] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009.

[6] Gianpiero Cabodi, Luciano Lavagno, Marco Murciano, Alex Kondratyev, and Yosinori Watanabe. Speeding-up heuristic allocation, scheduling and binding with sat-based abstraction/refinement techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 15(2), March 2010.

[7] Sheng Chen, Martin Erwig, and Eric Walkingshaw. A Calculus for Variational Programming. In *European Conf. on Object-Oriented Programming (ECOOP)*, volume 56 of *LIPIcs*, pages 6:1–6:26, 2016.

[8] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[9] Niklas Een, Alan Mishchenko, and Nina Amla. A single-instance incremental sat formulation of proof- and counterexample-based abstraction.

[10] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[11] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

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

[13] Martin Erwig and Eric Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV (GTTSE 2011), Revised and Extended Papers*, volume 7680 of *LNCS*, pages 55–99, 2013.

[14] Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying smt in symbolic execution of microcode. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, page 121–128, Austin, Texas, 2010. FMCAD Inc.

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

[16] Vijay Ganesh, Charles W. O'Donnell, Mate Soos, Srinivas Devadas, Martin C. Rinard, and Armando Solar-Lezama. Lynx: A programmatic sat solver for the rna-folding problem. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, page 143–156, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] James Garson. Modal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2018 edition, 2018.

[18] Spencer Hubbard and Eric Walkingshaw. Formula Choice Calculus. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 49–57. ACM, 2016.

[19] Stephen Cole Kleene. *Introduction to metamathematics*. Ishi Press, 1968.

[20] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. Technical Report 2016-01, TU Braunschweig, 2016.

[21] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Int. Conf. on Aspect-Oriented Software Development*, pages 191–202, 2011.

[22] Inês Lynce and João Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, page 136–141, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, May 1999.

[24] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. Anomaly Detection and Explanation in Context-Aware Software Product Lines. pages 18–21, 2017.

[25] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. pages 643–654, 2016.

[26] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental sat. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 206–218, Cham, 2014. Springer International Publishing.

[27] Nicholas Rescher. *Many-Valued Logic*. New York: Mcgraw-Hill, 1969.

[28] Ofer Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In Tiziana Margaria and Tom Melham, editors, *Correct Hardware Design and Verification Methods*, pages 58–70, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[29] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[30] JOM Silva and Karem A Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 152–161. IEEE, 1997.

[31] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. pages 47–60, 2011.

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

[33] Frank van Harmelen, Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation*. Elsevier Science, San Diego, CA, USA, 2007.

[34] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. pages 1–13, 2018.

[35] Eric Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013. http://hdl.handle.net/1957/40652.

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

[37] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. Satire: A new incremental satisfiability engine. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 542–545, New York, NY, USA, 2001. Association for Computing Machinery.

APPENDICES

Appendix A: Redundancy

This appendix is inoperable.