AN ABSTRACT OF THE DISSERTATION OF

Jeffrey M. Young for the degree of Doctor of Philosophy in Computer Science

presented on May 28, 2021.

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 result from accidental complexity produced by solving a problem that is *variational* without the concept of

variation, similar to problematic use of GOTO statements which occur without the concept of looping, and thus without WHILE loop constructs. 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 May 28, 2021
Commencement June 2021

Doctor of Philosophy dissertation of Jeffrey M. Young presented on May 28, 2021.

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

do it

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

TABLE OF CONTENTS

LIST OF FIGURES

Todo list

Chapter 1: Introduction

Controlling complexity is a central goal of any programming language, especially as software written in that language grows. The burgeoning field of *variation theory* and *variational programming* [30, 29, 37, 21, 67] attempt to control a kind of complexity which is induced into software when many *similar yet distinct* kinds of the same software 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[3], "No theory is complete without proof that the ideas work in practice"; this is the 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 variational data structures 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 [11] 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 [22], SAT solvers based on conflict-driven clause learning (CDCL) [46, 58, 9] have been able to solve boolean formulae with millions variables quickly enough for use in real-world applications [63]. Leading to their proliferation into several fields of scientific inquiry ranging from software engineering to Bioinformatics [45, 34].

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 [59, 57, 68, 13, 27, 25, 32] require solving a set of related SAT problems [57, 59, 25]. To take just one example, software product-lines (SPL) utilizes SAT solvers for a diverse range of analyses including: automated feature model analysis [10, 33, 62], feature model sampling [49, 64], anomaly detection [2, 40, 47], and dead code analysis [61].

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) [57, 68] 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 [26].

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 contain *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 introduces the essential aspects of variation using propositional logic and in the process presents the first instance of a *variational system recipe* to construct a *variation-aware* system using a non-variational version of that system. Several variational concepts are defined and formalized which are used throughout the thesis, such as *variant*, *configuration* and *variational artifact*. Lastly, the section proves theorems that are central to proof of the soundness of variational satisfiability solving.

3. Chapter 4 (*Variational Satisfiability Solving*) makes the central contribution of the thesis. In this chapter we define the general approach and architecture of a variational satisfiability solving. The general approach is the second presentation of

the aforementioned recipe; in this case using a SAT solver rather than propositional logic. This section provides a rationale for this design and makes several important contributions:

(a) An operational semantics of variational satisfiability solving. A variational SAT problem is a description of the problem in variational propositional logic that is translated to an incremental SAT program which is suitable for execution on an incremental SAT solver.

(b) A formal definition of concepts such as a *variational core* which are transferable to domains other than SAT. Variational cores are instances of var

(c) A definition of a *variational transpiler*. The transpiler is defined as a variational fold which is the basis for the performance gains presented in the thesis. The folding algorithm is defined in three phases to ensure that non-variational terms are shared across SAT problems and thus redundant computation is mitigated.

(d) A definition of a variational output that is returned to the user. The output presents several unique challenges that must be overcome while still being useful for the end user. We present and consider these concerns and provide a salable solution.

4. Chapter 5 (*Variational Satisfiability-Modulo Theory Solving*) extends the variational solving algorithm to consider SMT theories and propositions which include numeric values such as Integers and Reals in addition to Booleans. We present the requisite extensions to the variational propositional logic, the variational tran-

spiler and solving algorithm, and extend the output to support types other than just Booleans. While these extensions are relatively straightforward, complete support of SMT theories would need to include `BitVector` and `Arrays`. Supporting these theories is not straightforward, thus the chapter concludes by outlining the problem space for adding these features.

5. Chapter 6 (*Case Studies*) The central project of this thesis is to evaluate the ideas of variational programming in satisfiability solving. Having defined and constructed a variational SAT and SMT solver in the previous chapters this chapter empirically evaluates the solvers. The first section demonstrates performance improvements in variational SAT's intended use case. The second section present variational versions of classic SAT problems such as: ... and serves as a tutorial for identifying, writing, and solving variational SAT problems with a variational SAT solver.

fill after making examples

6. Chapter 7 (*Related Work*) is split into two sections. First, this thesis is situated into a lineage of recent variational-aware systems, thus this section collects this research and provides a comparison of our method to create a variational-aware system with previous methods. Second, this work is related to numerous SAT solvers that attempt to reuse information, solve sets of SAT problems and implement incremental SAT solving. We situate this work in the context of these solvers and compare their methods.

7. Chapter 8 (*Conclusion and Future Work*) summarizes the contributions of the thesis and relates the work to the central project of the thesis. In addition to the

not sure how to write this one yet

conceptual point, numerous areas of future work are discussed; from further variational extensions to faster implementation strategies and novel application domains. ...

Chapter 2: Background

This section provides background on SAT and incremental SAT solving. It is intended as a general introduction to these concepts. Specific techniques or algorithms are not discussed in detail. All descriptions follow the SMT-LIB2 [8] standard and describe incremental solvers as a black box eliding internal details of any specific solver which adheres to the standard.

2.1 SMTLIB2 and Satisfiability Solving

This section provides assumes knowledge of propositional logic, and provides background to satisfiability solving and SMTLIB2 (smtlib); the standardized language for interacting with SAT solvers. Following the notation from the many-valued logic community [54] we refer to propositional logic as C_2, which denotes a two-valued logic.

A satisfiability solver is a software system that solves the Boolean Satisfiability Problem [55]. One of the oldest problems in computer science[1] and famously NP-complete [22], the Boolean satisfiability problem is the problem of determining if a formula (sometimes called a sentence) in propositional logic has an assignment of Boolean values to variables, such that under substitution the formula evaluates to `T`. We formalize the problem and terms in the following definitions:

[1] see Biere et al. [12] for a complete history from the ancients, through to George Boole to the modern day.

Definition 2.1.1 (Model)**.** Given a formula in propositional logic: $f \in C_2$, which contains a set of Boolean variables vs. A model, m, is a set of assignments of Boolean values to variables in f such that f evaluates to $\mathtt{T}$, i.e., $m = \{(v \coloneqq b) \mid v \in vs, b \in \mathbb{B}\}$.

Corollary 2.1.0.1 (Validity)**.** *In propositional logic a formula or sentence is* valid *if it is true in* all *possible models [55]. That is, a valid formula or sentence is also a tautology.*

not sure if this is the right place

Definition 2.1.2 (Satisfiable)**.** Given a formula in propositional logic, f, which contains a set of Boolean variables vs. If there exists an assignment of variables to Boolean values such that f evaluates to $\mathtt{T}$, then we say f is *satisfiable*.

For example, we can show that the formula $good = (a \wedge b) \vee c$ is satisfiable with the model: $\{(a \coloneqq \mathtt{T}), (b \coloneqq \mathtt{T}), (c \coloneqq \mathtt{F})\}$, because $(\mathtt{T} \wedge \mathtt{T}) \vee \mathtt{F}$ results in $\mathtt{T}$. However, a formula such as $bad = (a \vee b) \wedge \mathtt{F}$ is not satisfiable as no assignment of $\mathtt{F}$ or $\mathtt{T}$ to the variables a and b would allow bad to evaluate to $\mathtt{T}$. With the preliminaries concepts we can now define the Boolean Satisfiability Problem:

Definition 2.1.3 (Boolean Satisfiability Problem)**.** Given a formula in propositional logic, f, determine if f is satisfiable.

While the formal definition of the Boolean Satisfiability Problem requires a formula in propositional logic, expressing a SAT problem in propositional logic can be cumbersome. Thus, modern satisfiability solvers programming languages to express SAT problems, communicate the problems to other people and dictate the problems to the solver. In recent years these programming languages have coalesced into a single standard via an international initiative called SMTLIB2.

The SMTLIB2 [8] standard formalizes a set of programming languages that define interactions with a SAT or SMT solver. The standard defines four languages, of which only two are used throughout this thesis: a *term* language; which defines a language for defining variables, functions and formulas in propositional and first-order logic. The *command* language; which defines a programming language to interact with the solver. The command language is used to add or remove formulas, query the solver for a model or check for satisfiability and other side-effectual interactions such as printing output.

For the remainder of this section we provide informal examples intended for a general audience and cover only the commands and concepts required for subsequent sections of this thesis. For a full language specification please see Barrett et al. [8].

define a listing language for smtlib

Consider this SMTLIB2 program which verifies peirce's law implies the law of excluded middle for propositional logic:

```
(declare-const a Bool)                        ;; variable declarations
(declare-const b Bool)
(define-fun ex-middle ((x Bool)) Bool         ;; excluded middle: x ∨ ¬x
  (or x
      (not x)))
(define-fun peirce ((x Bool) (y Bool)) Bool   ;; peirce's law: ((x → y) → x) → x
  (=>
    (=> (=> x y)
        x)
    x))
(define-fun peirce-implies-ex-middle () Bool
  (=> (peirce a b)
      (ex-middle a)))
(assert (not peirce-implies-ex-middle))       ;; add assertion
(check-sat)                                   ;; check SAT of all assertions
```

Comments begin with a semi-colon (;) and end at a new line. The program, and every SMTLIB2 program, is a sequence of *commands* (called *statements* in the programming language literature) that interact with the solver. For example, the above program consists of five commands, two variable declarations, a function definition, an assertion and a command to check satisfiability. Each command is formulated as an *s-expressions* [48] to simplify parsing. For our purposes, one only needs to understand that commands and functions are called by opening parentheses; the first element after the opening parenthesis is the name of the command or function symbol and every other element is an argument to that command. Thus (declare-const a Bool) is an s-expression with three elements that defines the C_2 variable a of *sort* (called *type* in programming language literature) Bool. The first element, declare-const is the command name, the second is the user defined name for the variable and the third is its sort. Similarly, the s-expression (and a b) passes the variables a and b to the function and which returns the conjunction of these two variables. Lastly, the function definition define-fun takes four arguments: the user defined name; peirce-is-ex-middle, an s-expression that defines argument names and their sorts; ((x Bool) (y Bool)), a return sort; Bool and the body of the function.

Internally, a compliant solver such as z3 [23] maintains an stack called the *assertion stack* that tracks user provided variable and formula declarations and definitions. The elements of the assertion stack are called *levels* and are sets of *assertions*. An assertion is a logical formula, a declaration of a sort, or a definition of a function symbol. In the example, both variable declarations and the peirce-is-ex-middle definition are included in the assertion set. Sets of assertions are placed on the stack via the assert command. The assert command takes a term as input[2], collects all associated definitions and declarations and places the assertion set on the assertion stack.

---

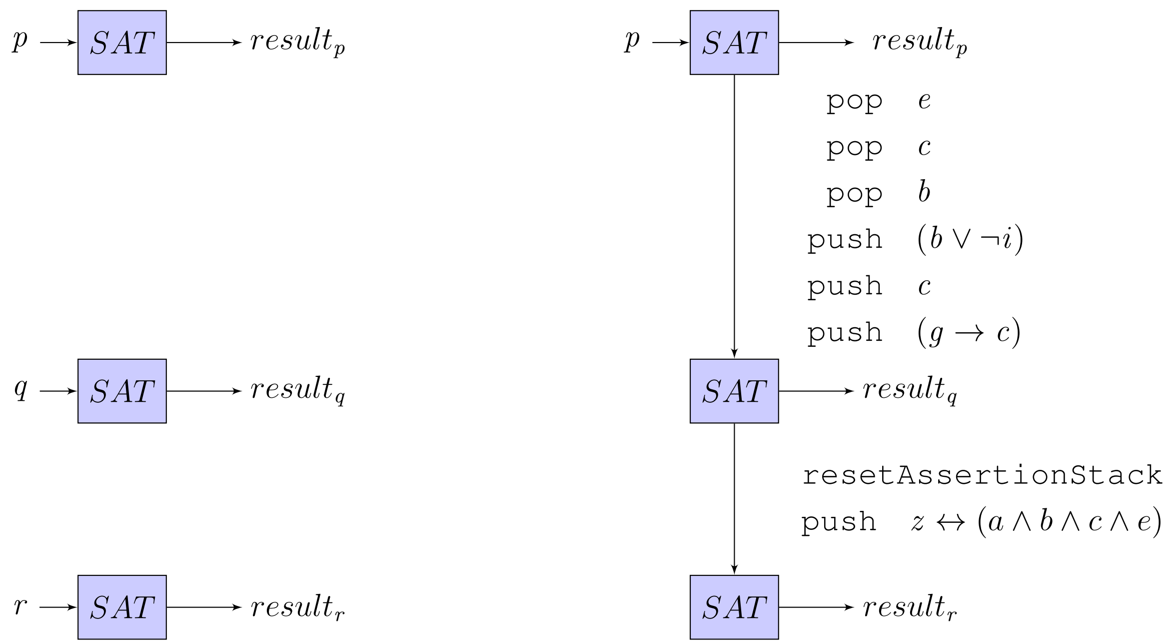
[2]by the standard this is a *well-sorted term* of type *Bool*, however we elide this description for simplicity

The example demonstrates a common verification pattern in SAT and SMT solving. In the example, we construct a constraint that asserts (not peirce-is-ex-middle) rather than peirce-is-ex-middle because we need to verify that peirce's law implies the law of excluded middle for all possible models. Had we elided the not then the first model which satisfied the theorem would be returned, thus providing a *single* model where the theorem holds. However, to prove the theorem we need to show that it holds *for all* possible models. The not negates the theorem thus asking the solver to discover a counter-example to the theorem. If such a model exists, then the solver has discovered a counter-example to the theorem. If no such model exists—that is UNSAT is returned by the solver—then the negated theorem always evaluates to `F` and thus the theorem always evaluates to `T` and hence is logically valid.

Satisfiability and logical validity are closely related. Conceptually, satisfiability attempts to find a model that solves the constraints of a formula, while logical validity tries to show that the formula's truth-value is independent of it's variables and thus the formula is tautological. Similarly, where satisfiability is concerned with solving constraints, validity is concerned with finding a proof. Thus it is common to negate a formula to *query* the solver to search for a counter-example, when no such model is found and UNSAT is returned, we can be sure that the negation of a formula which is always `F` is a formula which is always `T`, and thus is logically valid.

2.2 Incremental Satisfiability Solving

Suppose, we have three related propositional formulas that we desire to solve.



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

(b) Incremental procedure, reuse defined by POP and PUSH.

Figure 2.1

$$p = a \wedge b \wedge c \wedge e \quad q = a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c) \quad r = z \leftrightarrow (a \wedge b \wedge c \wedge e)$$

p is simply a conjunction of variables. In q, relative to p, we see that two variables are added, i, g, e is removed, and there are two new clauses: $(b \vee \neg i)$ and $(g \rightarrow c)$, both of which possibly affect the values of b and c. In r, the variables and constraints introduced in p are further constrained to a new variable, z.

Suppose one wants to find a model for each formula. Using a non-incremental SAT solver results in the procedure illustrated in Fig. 2.1a; where SAT solving is a batch process and no information is reused. Alternatively, a procedure using an incremental

SAT solver is illustrated in Fig. 2.1b; in this scenario, all formulas are solved by single solver instance where terms are programmatically added or removed from the solver.

The ability to add and remove terms is enabled by manipulating the *assertion stack*. The previous example demonstrated a single level on the stack, this example demonstrates three. The incremental interface provides two new commands: PUSH to create a new variable *scope* and add a level to the stack and POP to remove the level. The following program follows the procedure outlined in Fig. 2.1b and solves p, q and r:

```
(declare-const a Bool)          ;; variable declarations for p
(declare-const b Bool)
(declare-const c Bool)
(declare-const e Bool)
(assert a)                      ;; a is shared between p and q
(push)                          ;; solve p
  (assert e)
  (assert c)
  (assert b)
  (check-sat)                   ;; check-sat on p
(pop)                           ;; remove e, c, and b assertions
(push)                          ;; solve for q
  (declare-const i Bool)        ;; new variables
  (declare-const g Bool)
  (assert (or b (not i)))       ;; new clause
  (assert c)                    ;; re-add c
  (assert (=> g c))             ;; new clause
  (check-sat)                   ;; check sat of q
(pop)                           ;; i and g out of scope
(reset)                         ;; reset the assertion stack
(declare-const a Bool)          ;; variable declarations for r
(declare-const b Bool)
(declare-const c Bool)
(declare-const e Bool)
(declare-const z Bool)
(assert (= z (and a (and b (and c (and e ))))))
(check-sat)                     ;; check-sat on r
```

In this example we begin by defining p, we assert a outside of a new scope so that it can be reused for q. We reuse a by exploiting the conjunction of assertions per level on the assertion stack during a CHECK-SAT call. Had we asserted `(and a (and b (and c (and e))))` then we would not be able to reuse the assertion on a. The first PUSH command enters a new level on the assertion stack, the remaining variables are asserted and we issue a check-sat call. After the POP command, all assertions and declarations from the previous level are removed. Thus, after we solve q the variables i and g cannot be referenced as they are no longer in scope.

In an efficient process one would initially add as many *shared* terms as possible, such as a from p and then reuse that term as many times as needed. Thus, an efficient process should perform only enough manipulation of the assertion stack as required to reach the next SAT problem of interest from the current one. However, notice that doing so is not entirely straight forward; we were only able to reuse a from p in q because we defined p in a non-intuitive way by utilizing the internal behavior of the assertion stack. This situation is exacerbated by SAT problems such as r where due to the equivalence we were forced to completely remove everything on the stack in order to construct r. Thus incremental SAT solvers provide the primitive operations required to solve related SAT problems efficiently, yet writing the SMTLIB2 program to solve the set efficiently is not straightforward.

this is motivation but seems like a good time to make this point?

Chapter 3: Variational Propositional Logic

In this chapter, we present the logic of variational satisfiability problems. The logic is a conservative extension of classic two-valued logic (C_2) with a *choice* construct from the choice calculus [29, 66], 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 chapter defines the syntax and semantics of VPL and concludes with a set of definitions, lemmas and theorems for the logic.

3.1 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 thesis.

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

(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.

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

(c) VPL equivalence laws.

Figure 3.1: Formal definition of VPL.

3.2 Semantics

Conceptually, a variational formula represents several propositional logic formulas at once, which can be obtained by resolving all of the choices. For software product-line researchers, it is useful to think of VPL as analogous to `#ifdef`-annotated C_2, where choices correspond to a disciplined [43] application of `#ifdef` annotations. From a logical perspective, following the many-valued logic of Kleene [39, 54], 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 [35], because a choice *waits* until there is enough information in an external environment 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.

3.3 Formalisms

Having defined the syntax and semantics of VPL the rest of this chapter will define useful functions and properties of VPL formulas. Lastly, we conclude the chapter with an example of encoding a set of C_2 formulas to a single VPL formula.

We begin with useful functions to retrieve interesting aspects of VPL formulas.

Definition 3.3.1 (Dimensions)**.** Given a formula $f \in$ VPL, let $Dimensions(f)$ be the set of unique dimensions in the formula. Thus, $Dimensions(f) = \{D \mid D \in f\}$.

For example, $Dimensions(A\langle p, q\rangle \wedge B\langle r, s\rangle) = \{A, B\}$ and $Dimensions(A\langle p, q\rangle \wedge A\langle r, s\rangle)$ = $\{A\}$. Similarly we define a notion of *cardinality* over VPL formulas.

Definition 3.3.2 (Dimension-cardinality)**.** The dimension-cardinality or d-cardinality of a formula $f \in$ VPL is the cardinality of the set of unique dimensions in a formula. We use the following notation as shorthand: $|f|_D = |Dimensions(f)|$.

Similarly to $Dimensions$ another useful function is $Variants$:

Definition 3.3.3 (Variants)**.** Given a formula $f \in$ VPL, let $Variants(f)$ be the set of all possible *plain variants* of f. Thus, $Variants(f) = \{\, v \mid \exists C.\ v = [\![f]\!]_C,\ v \in C_2\,\}$

Using $Dimensions$ we can now define a more precise property on configurations.

Definition 3.3.4 (Valid Configuration)**.** We say a configuration C is valid with respect to some formula $f \in$ VPL iff $Dom(C)\ \cap\ Dimensions(f)\ \neq \varnothing$.

One may think of a valid configuration as a total configuration with *nothing extra*. For example, the configuration $C = \{(A, \mathsf{true}), (B, \mathsf{false}), (E, \mathsf{true})\}$ is total with respect to the formula $f = A\langle p, q\rangle \wedge B\langle r, s\rangle$ because C eliminates all choices in f. However C is not valid with respect to f because $Dom(C) \cap Dimensions(f) = \{E\}$, and thus in this case C contains one extra binding for E, than needed with respect to f.

Definition 3.3.5 (Variant)**.** Let f be a variant of a formula e iff there exists some valid, total configuration C such that $[\![e]\!]_C = f$.

Because variants are defined only with total configurations, all variants cannot contain choices and are hence called *plain*:

Definition 3.3.6 (Plain formula)**.** For any formula $e \in$ VPL, let e be plain iff $Dimensions(e) = \varnothing$

Lemma 3.3.1 (Variants are plain)**.** *By* **??** *and the fact that variants are found via total configurations*

3.4 Background example

To demonstrate the application of VPL and conclude the chapter, we encode the incremental example from Chapter 2. Our goal is to construct a single variational formula that encodes the related plain formulas p, q, r. Ideally, this variational formula should maximize sharing among the plain formulas in order to avoid redundant analyses during a variational solving. We reproduce the formulas below for the convenience:

$$p = a \wedge b \wedge c \wedge e \quad q = a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c) \quad r = z \leftrightarrow (a \wedge b \wedge c \wedge e)$$

section on encoding problem

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. Unfortunately, the process of globally minimizing a variational formula in this way is hard[1] since 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 P, Q, R to represent the respective variants. Unique portions of each variant will be nested into the left alternative so that the unique portion is considered when the dimension is bound to truein the configuration.

We begin by combining p and r since the differences[2] 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 p and r is the addition of z and thus we wrap the leaf in a choice with dimension R. This yields the following variational formula.

$$f_{pr} = R\langle z, \mathtt{T}\rangle \leftrightarrow (a \wedge b \wedge c \wedge e)$$

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

[1]. We hypothesize that it is equivalent to BDD minimization, which is NP-complete, but the equivalence has not been proved; see [67].

[2]There are many ways to assess the difference of two formulas. For now the reader may consider it reducible to the Levenshtein distance of two strings [42]. We return to this discussion in ...

Next we combine f_{pr} with q to obtain a variational formula that encodes the propositional formulas p, q, r. There are two sub-trees that must be wrapped in choices. First, we must encode the difference between $b \vee \neg i$ from q and b. Second, we must ensure synchronization and thus use the same dimension to encode the difference between $g \rightarrow c$ and e. Thus the resulting variational formula is:

$$f = R\langle z, \mathtt{T}\rangle \leftrightarrow (a \wedge Q\langle b \vee \neg i, b\rangle \wedge c \wedge Q\langle g \rightarrow c, e\rangle)$$

Now that we have constructed the variational formula we need to ensure that it encodes all variants of interest and nothing else. Notice that only 2 dimensions are used to encode 3 variants, because $|f|_D$ = 2 we have are 4 possible variants and thus one extra variant. We can observe this by enumerating the variants and possible configurations:

$$\begin{aligned}
p &= \mathtt{T} \leftrightarrow (a \wedge b \wedge c \wedge e) & C = \{(R, \mathsf{false}), (Q, \mathsf{false})\} \\
q &= \mathtt{T} \leftrightarrow (a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c)) & C = \{(R, \mathsf{false}), (Q, \mathsf{true})\} \\
r &= z \leftrightarrow (a \wedge b \wedge c \wedge e) & C = \{(R, \mathsf{true}), (Q, \mathsf{false})\} \\
\mathit{extra} &= z \leftrightarrow (a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c)) & C = \{(R, \mathsf{true}), (Q, \mathsf{true})\}
\end{aligned}$$

Notice the extra variant and that p and q are only recovered through equivalency laws from propositional logic. While it is undesirable that there exists extra variants, the important constraint: $\mathit{fs} = \{v \mid v \in \{p, q, r\}\}$, $fs \subseteq \mathit{Variants}(f)$ is satisfied. We'll return to the case of extra variants in the next chapter as a variational SAT solver must only solve variants of interest.

maybe superfluous?

Chapter 4: Variational Satisfiability Solving

In this section, we present our algorithm for variational satisfiability solving. Section 4.1 provides an overview of the algorithm and introduces the notion of *variational models* as solutions to variational satisfiability problems. **??** provides the formal specification.

4.1 General Approach

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

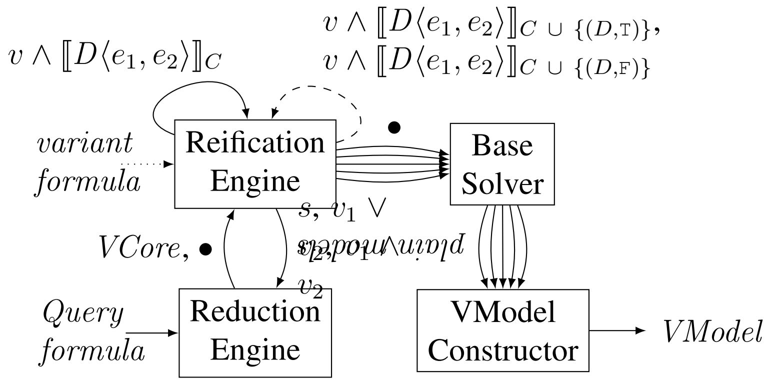


Figure 4.1: System overview of the variational solver.

the result of solving all variants of the original VPL formula.

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

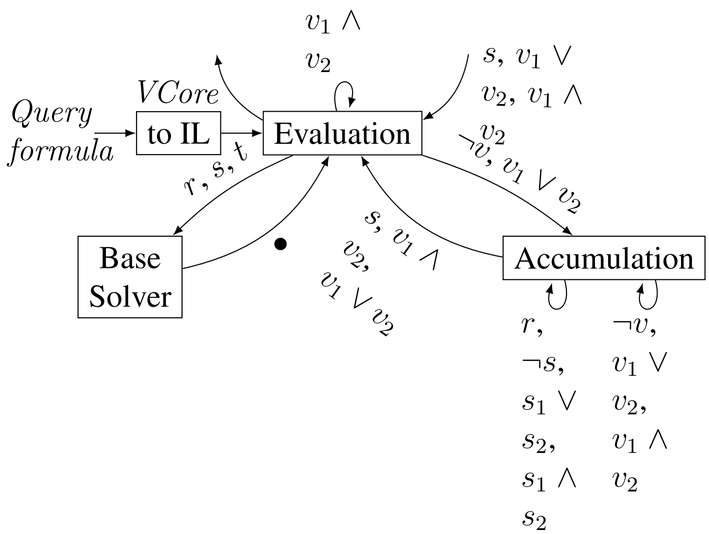


Figure 4.2: Overview of the reduction engine.

it onto the assertion stack, and create a symbolic reference for $(p \wedge \neg q)$.

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

[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 [52] can be used to implement variational satisfiability solving.

by manipulating the configuration. In **??**, we formalized a configuration as a function $D \to \mathbb{B}$, which we encode in the solver as a set of tuples $\{D \times \mathbb{B}\}$. Fig. 4.1 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 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

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

Figure 4.3: Possible plain models for variants of [do]. . . .

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

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

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 vc_r, and maintain a special variable called $\_Sat$ to track which configurations are satisfiable.

For example, consider the query formula [plug]. . . 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

remove references the linux example

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 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*.

fix

4.2 Formalization

In this subsection we formalize variational SAT solving by specifying the semantics of the *accumulation* and *evaluation* phases of the variational solver, as well as the semantics of processing the variational core, which we call *choice removal*. Variational SAT solving assumes the existence of an underlying incremental SAT solver, which we refer to as the *base solver*.

The variational solver interacts with the base solver via several primitive operations.

$$\begin{array}{rllll} \texttt{Not} : & (\Delta, s) & \rightarrow & (\Delta, s) & \textit{Negate a symbolic value} \\ \texttt{And} : & (\Delta, s, s) & \rightarrow & (\Delta, s) & \textit{Conjunction of symbolic values} \\ \texttt{Or} : & (\Delta, s, s) & \rightarrow & (\Delta, s) & \textit{Disjunction of symbolic values} \\ \texttt{Var} : & (\Delta, r) & \rightarrow & (\Delta, s) & \textit{Create symbolic value based on a variable} \\ \texttt{Assert} : & (\Gamma, \Delta, s) & \rightarrow & \Gamma & \textit{Assert a symbolic value to the solver} \\ \texttt{GetModel} : & (\Gamma, \Delta) & \rightarrow & m & \textit{Get a model for the current solver state} \end{array}$$

Figure 4.5: Assumed base solver primitive operations.

In our semantics, we simulate the effects of these primitive operations by tracking their effects on two stores. The *accumulation store* Δ tracks values cached during accumulation by mapping IL terms to symbolic references. The *evaluation store* Γ tracks the symbolic references that have been sent to the base solver during evaluation.

Primitives Fig. 4.5 lists a minimal set of primitive operations that the base solver is assumed to support. These primitive operations define the interface between the base solver and the variational solver.

The primitive operations can be roughly grouped into two categories: The first four operations, consisting of the logical operations `Not`, `And`, and `Or`, plus the `Var` operation, are used in the accumulation phase and are concerned with creating and maintaining symbolic references that may stand for arbitrarily complex subtrees of the original formula. These operations simulate caching information in the base solver. The final two operations, `Assert` and `GetModel`, are used in the evaluation phase and simulate pushing new assertions to the base solver and obtaining a satisfiability model based on the current solver state, respectively.

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

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

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

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

$$\begin{aligned} \texttt{Var}(\Delta_0, p) &= (\Delta_1, s_p) \\ \texttt{Var}(\Delta_1, q) &= (\Delta_2, s_q) \\ \texttt{Not}(\Delta_2, s_q) &= (\Delta_3, s_q') \\ \texttt{And}(\Delta_3, s_p, s_q') &= (\Delta_4, s_{pq}) \end{aligned}$$

The accumulation store tracks what information is associated with each symbolic reference. The store must therefore be threaded through the calls to each primitive operation so that subsequent operations have access to existing definitions and can produce a new, updated store. For example, the final store produced by the above example contains the following mappings from IL terms to symbolic references, $\Delta_4 = \{(p, s_p), (q, s_q), (\neg s_q, s'_q), (s_p \wedge s'_q, s_{pq})\}$.

When comparing the smtlib notation to our formalization, observe that each use of declare-const corresponds to a use of the `Var` primitive, while the declare-fun line in smtlib may potentially expand into several primitive operations in our formalization. For the evaluation primitives, the `Assert` operation corresponds to an smtlib assert call, while the `GetModel` operation corresponds roughly to an smtlib check-sat call, which retrieves a model for the current set of assertions on the stack. However, the exact semantics of check-sat depends on the base solver in use. For example, given the plain formula $p = a \vee b \vee c$, z3 returns only a minimal satisifiable model, such as $\{b = \mathtt{T}\}$, providing no values for the other variables in the formula. To normalize this behavior across solvers, we instead consider `GetModel` equivalent to check-sat followed by a get-value call call for every variable in the query formula, yielding a complete model. For example, a complete model for p would be $\{a = \mathtt{F}, b = \mathtt{T}, c = \mathtt{F}\}$.

Finally, in Fig. 4.6 we define wrapped versions of the primitive operations used in accumulation. These wrapper functions first check to see whether a symbolic reference for the given IL term exists already in the accumulation store, and if so, returns it without changing the store. Otherwise, it invokes the corresponding primitive operation to generate and return the new symbolic reference and updated store.

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

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

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

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

Figure 4.6: Wrapped accumulation primitive operations.

Accumulation The accumulation phase is formally specified in Fig. 4.7 as a relation of the form $(\Delta, v) \mapsto (\Delta', v')$. Accumulation replaces plain subterms of the formula with references to symbolic values, wherever possible. The replacement of subterms by symbolic references is achieved by the first four rules in the figure. In the A-REF rule, a variable reference is replaced by a symbolic reference by invoking the wrapped version of the `Var` primitive, which returns the corresponding symbolic reference or generates a new one, if needed. The A-NOT-S, A-AND-S, and A-OR-S rules all similarly replace an IL term by a symbolic reference by invoking the corresponding wrapped primitive operation. These rules all require that their subterms completely reduce to symbolic references, as illustrated by the premise $(\Delta, v) \mapsto (\Delta', s)$ in the A-NOT-S rule, otherwise the primitive operation cannot be invoked.

However, not all subterms can be completely reduced to symbolic references. In particular, variational subterms—subterms that contain one or more choices within them—

$$\frac{\underline{\texttt{Var}}(\Delta, r) = (\Delta', s)}{(\Delta, r) \mapsto (\Delta', s)} \text{ A-REF}$$

$$\frac{(\Delta, v) \mapsto (\Delta', s) \qquad \underline{\texttt{Not}}(\Delta', s) = (\Delta'', s')}{(\Delta, \neg v) \mapsto (\Delta'', s')} \text{ A-NOT-S}$$

$$\frac{\begin{array}{c}(\Delta, v_1) \mapsto (\Delta_1, s_1) \\ (\Delta_1, v_2) \mapsto (\Delta_2, s_2) \qquad \underline{\texttt{And}}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)\end{array}}{(\Delta, v_1 \wedge v_2) \mapsto (\Delta_3, s_3)} \text{ A-AND-S}$$

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

$$\frac{}{(\Delta, D\langle e_1, e_2\rangle) \mapsto (\Delta, D\langle e_1, e_2\rangle)} \text{ A-CHC} \qquad \frac{(\Delta, v) \mapsto (\Delta', v')}{(\Delta, \neg v) \mapsto (\Delta', \neg v')} \text{ A-NOT-V}$$

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

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

Figure 4.7: Accumulation inference rules.

cannot be accumulated to a symbolic reference. The A-CHC rule prevents accumulation under a choice. The A-NOT-V, A-AND-V, and A-OR-V rules are congruence rules that recursively apply accumulation to subterms. Although not explicitly stated in the premises, it is assumed that these A-*-V rules apply only if the corresponding A-*-S rule does not apply, that is, when at least one of the subterms does not reduce completely to a symbolic reference.

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

To illustrate the semantics of accumulation, consider the plain formula $g = a \vee (a \wedge b)$ with an initial accumulation store $\Delta = \varnothing$. The A-OR-S rule matches the root $\vee$ connective with $v_1 = a$ and $v_2 = a \wedge b$. Since v_1 is a reference, the A-REF rule applies, generating a new symbolic reference s_a and returning the store $\Delta_1 = \{(a, s_a)\}$. Processing v_2 requires an application of the A-AND-S rule with $v'_1 = a$ and $v'_2 = b$, both of which require another application of the A-REF rule. For v'_1, the variable a is found in the store returning s_a, while for v'_2, a new symbolic reference s_b is generated and added to the resulting store $\Delta_2 = \{(a, s_a), (b, s_b)\}$. Since both the left and right sides of v_2 reduce to a symbolic reference, the `And` primitive is invoked, yielding a new symbolic reference s_{ab} and the store $\Delta_3 = \{(a, s_a), (b, s_b), (a \wedge b, s_{ab})\}$. Finally, since both the left and right sides of the original formula g reduce to symbolic references, the `Or` primitive is invoked yielding the final symbolic reference s_g and the final accumulation store $\Delta_4 = \{(a, s_a), (b, s_b), (s_a \wedge s_b, s_{ab})(s_a \vee s_{ab}, s_g)\}$.

When a formula contains choices, all of the plain subterms surrounding the choices

$$\frac{(\Delta, v) \mapsto (\Delta', v') \qquad (\Gamma, \Delta', v') \rightarrowtail (\Gamma', \Delta'', v'')}{(\Gamma, \Delta, v) \rightarrowtail (\Gamma', \Delta'', v'')} \text{ E-ACC} \qquad \frac{\texttt{Assert}(\Gamma, \Delta, s) = \Gamma'}{(\Gamma, \Delta, s) \rightarrowtail (\Gamma', \Delta, \bullet)} \text{ E-SYM}$$

$$\frac{}{(\Gamma, \Delta, D\langle e_1, e_2\rangle) \rightarrowtail (\Gamma, \Delta, D\langle e_1, e_2\rangle)} \text{ E-CHC} \qquad \frac{}{(\Gamma, \Delta, v_1 \vee v_2) \rightarrowtail (\Gamma, \Delta, v_1 \vee v_2)} \text{ E-OR}$$

$$\frac{(\Gamma, \Delta, v_1) \rightarrowtail (\Gamma_1, \Delta_1, \bullet) \qquad (\Gamma_1, \Delta_1, v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_2')}{(\Gamma, \Delta, v_1 \wedge v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_2')} \text{ E-AND-L}$$

$$\frac{(\Gamma, \Delta, v_1) \rightarrowtail (\Gamma_1, \Delta_1, v_1') \qquad (\Gamma_1, \Delta_1, v_2) \rightarrowtail (\Gamma_2, \Delta_2, \bullet)}{(\Gamma, \Delta, v_1 \wedge v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_1')} \text{ E-AND-R}$$

$$\frac{(\Gamma, \Delta, v_1) \rightarrowtail (\Gamma_1, \Delta_1, v_1') \qquad (\Gamma_1, \Delta_1, v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_2')}{(\Gamma, \Delta, v_1 \wedge v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_1' \wedge v_2')} \text{ E-AND}$$

Figure 4.8: Evaluation inference rules.

are accumulated to symbolic references, but choices remain in place and their alternatives are not accumulated. For example, consider the variational formula $g' = (a \vee (a \wedge b)) \vee D\langle a, a \wedge b\rangle \wedge (a \vee (a \wedge b))$ which contains two instances of g as subterms. The formula g' accumulates to the variational core $s_g \vee D\langle a, a \wedge b\rangle \wedge s_g$ with the same final store Δ_4 produced when accumulating g alone. Note that the each instance of g in g' was reduced to the same symbolic reference s_g and the alternatives of the choice were not reduced.

Evaluation The evaluation phase is formally specified in Fig. 4.8 as a relation of the form $(\Gamma, \Delta, v) \rightarrowtail (\Gamma', \Delta', v')$, where an evaluation store Γ represents the base solver's state. The E-ACC and E-SYM rules are the heart of evaluation: the E-ACC rule enables accumulating subterms during evaluation, while the E-SYM rule sends a fully accumulated subterm to the base solver. Evaluation cannot occur under choices or un-accumulated disjunctions (i.e. disjunctions that contain choices), as seen in the E-CHC and E-OR rules, but can occur under un-accumulated conjunctions, as reflected by the three E-AND* rules. This will be explained in more detail below.

When a subterm is sent to the base solver by E-SYM, it is replaced by the unit value $\bullet$ and the evaluation store Γ is updated accordingly. Conceptually, the evaluation store represents the internal state of the underlying solver (e.g. z3's internal state), but we model it formally as the set of assertions that have been sent to the solver. For example, given the accumulation store $\Delta = \{(a, s_a), (b, s_b), (s_a \wedge s_b, s_{ab})\}$, the assertion $\texttt{Assert}(\{\}, \Delta, s_a)$ yields $\{s_a\}$ and subsequent assertions add more elements to this set, for example, $\texttt{Assert}(\{s_a\}, \Delta, s_{ab}) = \{s_a, s_{ab}\}$. The assertions sent to a SAT solver are implicitly conjuncted together, which is why partially accumulated conjunctions may still be evaluated, but partially accumulated disjunctions may not. Such disjunctions are instead handled during choice removal using back-tracking.

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

Consider evaluating the formula $g = (a \vee b) \wedge D\langle a, c\rangle$ with initially empty stores.

We start by applying accumulation using the E-ACC rule, yielding the intermediate term $g' = s_{ab} \land D\langle a, c\rangle$ with the accumulation store $\Delta = \{(a, s_a), (b, s_b), (s_a \lor s_b, s_{ab})\}$. We then apply E-AND-L to g', which sends the left subterm s_{ab} to the base solver via the E-SYM rule, and the right side will be unevaluated via the E-CHC rule. Ultimately, evaluation yields the expression $D\langle a, c\rangle$ with accumulation store Δ and evaluation store $\{s_{ab}\}$.

Choice removal The main driver of variational solving is the choice removal phase, which is formally specificed in Fig. 4.9 as a relation of the form $(C, \Gamma, \Delta, M, z, v) \Downarrow M'$. The main role of choice removal is to relate an IL term v to a variational model M'. However, to do this requires several pieces of context including a configuration C, an evaluation store Γ, an accumulation store Δ, an initial variational model M, and an evaluation context z. The two stores have been explained earlier in this subsection, and variational models are explained at the end of Section 4.1. We explain configurations and evaluation contexts in the context of the relevant rules below.

The C-EVAL rule states that v fully evaluates to $\bullet$, then we can get the current model from the base solver using the `GetModel` primitive and update our variational model. We use the operation `Combine` to perform the variational model update operation described in Section 4.1. The rest of the choice removal rules are structured so that C-EVAL will be invoked once for every variant of the variational core so that the final output will be a variational model that encodes the solutions to every variant of the original formula.

The next three rules concern choices and are the heart of choice removal. These rules make use of a *configuration* C, which maps dimensions to Boolean values (encoded as

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

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

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

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

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

$$\frac{(\Delta, \neg s) \mapsto (\Delta', s') \qquad (C, \Gamma, \Delta, M, z, s') \Downarrow M'}{(C, \Gamma, \Delta, M, \neg\cdot :: z, s) \Downarrow M'} \text{ C-NOT-IN}$$

$$\frac{(C, \Gamma, \Delta, M, \cdot \wedge v_2 :: z, v_2) \Downarrow M'}{(C, \Gamma, \Delta, M, z, v_1 \wedge v_2) \Downarrow M'} \text{ C-AND}$$

$$\frac{(C, \Gamma, \Delta, M, s \wedge \cdot :: z, v) \Downarrow M'}{(C, \Gamma, \Delta, M, \cdot \wedge v :: z, s) \Downarrow M'} \text{ C-AND-INL}$$

$$\frac{(\Delta, s_1 \wedge s_2) \mapsto (\Delta', s_3) \qquad (C, \Gamma, \Delta, M, z, s_3) \Downarrow M'}{(C, \Gamma, \Delta, M, s_1 \wedge \cdot :: z, s_2) \Downarrow M'} \text{ C-AND-INR}$$

$$\frac{(C, \Gamma, \Delta, M, \cdot \vee v_2 :: z, v_2) \Downarrow M'}{(C, \Gamma, \Delta, M, z, v_1 \vee v_2) \Downarrow M'} \text{ C-OR}$$

$$\frac{(C, \Gamma, \Delta, M, s \vee \cdot :: z, v) \Downarrow M'}{(C, \Gamma, \Delta, M, \cdot \vee v :: z, s) \Downarrow M'} \text{ C-OR-INL}$$

$$\frac{(\Delta, s_1 \vee s_2) \mapsto (\Delta', s_3) \qquad (C, \Gamma, \Delta, M, z, s_3) \Downarrow M'}{(C, \Gamma, \Delta, M, s_1 \vee \cdot :: z, s_2) \Downarrow M'} \text{ C-OR-INR}$$

Figure 4.9: Choice removal inference rules

a set of pairs). The configuration tracks which dimensions have been selected and how to ensure that all choices in the same dimension are synchronized. Whenever a choice $D\langle e_1, e_2\rangle$ is encountered during choice removal, we check C to determine what to do. In C-CHC-T, if $(D, \mathsf{true}) \in C$, then the first alternative of the dimension has already been selected, so choice removal proceeds on e_1. Similarly, in C-CHC-F, if $(D, \mathsf{false}) \in C$, the right alternative has been selected, so choice removal proceeds on e_2. In C-CHC, if $D \notin \mathit{dom}(C)$, then the dimension has not yet been selected, so we recursively apply choice removal to both e_1 and e_2, updating C accordingly in each case. Observe that we use the same accumulation store, evaluation store, and evaluation context for each alternative. This simulates a backtracking point in the solver, where we first solve e_1, then reset the state of the solver to the point where we encountered the choice and solve e_2. Only the variational model, which is threaded through the solution of both e_1 and e_2, is maintained to accumulate the results of solving each alternative.

The final eight rules apply choice removal to the logical operations. These rules make heavy use of an *evaluation context* z that keeps track of where we are in a partially evaluated IL term during choice removal. Evaluation contexts are defined as a zipper data structure [38] over IL terms, given by the following grammar.

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

An evaluation context z is like a breadcrumb trail that enables focusing on a subterm within a partially evaluated IL term while also keeping track of work left to do. The empty context $\top$ indicates the root of the term. The other cases in the grammar prepend a "crumb" to the trail. The crumb $\cdot \neg$ focuses on the subterm within a negation, $\cdot \wedge v$ focuses on the left subterm within a conjunction whose right subterm is v, and $v \wedge \cdot$ focuses on the right subterm of a conjunction whose left subterm has already been reduced to s. The cases for disjunction are similar to conjunction.

As an example, consider the IL term $\neg(a \vee b) \wedge c$. When evaluation is focused on a, the evaluation context will be $\cdot \vee b :: \neg\cdot :: \cdot \wedge c :: \top$, which states that a exists as the left child of a disjunction whose right child is b, which is inside a negation, which is the left child of a conjunction whose right child is c. The b and c terms captured in the context are subterms of the original term that must still be evaluated. During choice removal, IL terms are evaluated according to a left-to-right, post-order traversal; as IL subterms are evaluated they are replaced by symbolic references via accumulation. When evaluation is focused on b, the context will be $s_a \vee \cdot :: \neg\cdot :: \cdot \wedge c :: \top$, where s_a is the symbolic reference produced by accumulating the variable a. When evaluation is eventually focused on c, the evaluation context will be simply $s_{ab} \wedge \cdot :: \top$ since the entire subtree $\neg(a \vee b)$ on the left side of the conjunction will have been accumulated to the symbolic reference s_{ab}.

The C-NOT, C-AND, and C-OR rules define what to do when encountering a logical operation for the first time. In C-NOT, we focus on the subterm of the negation, while in C-AND and C-OR, we focus on the left child while saving the right child in the context. The C-AND-INL and C-OR-INL rules define what to do when *finished* processing the left child of the corresponding operation. A fully processed child have been accumulated to a symbolic reference s. At this point, we move the s into the evaluation context and shift focus to the previously saved right child of the logical operation. Finally, the

C-Not-In, C-And-InR, and C-Or-InR rules define what to do when finished processing all children of a logical operation. At this point, all children will have been reduced to symbolic references so we can accumulate the entire subterm and apply choice removal to the result. For example, in C-And-InR, we have just finished processing the right child to s_2 and we previously reduced the left child to s_1, so we now accumulate $s_1 \vee s_2$ to s_3 and proceed from there.

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

Chapter 5: Variational Satisfiability-Modulo Theory Solving

Chapter 6: Case Studies

Chapter 7: Related Work

Related work has been discussed throughout the previous sections. This section collects related work that was not previously covered. To my knowledge, this work is the first to translate the specific ideas of a nanopass architecture, and a variational compiler, to the SAT/SMT domains. Furthermore, at time of this writing this work is the first to investigate variational concurrency.

This work is most similar to [65], which also constructs a SAT solver that exploits shared terms and prevents redundant computation. However, the projects differ in important ways. Visser et al.'s solver is oriented for program analysis and does not use incremental SAT solving. Rather, it uses heuristics to find canonical forms of sliced programs, and caches solver results on these canonical forms in a key-value store [41]. In contrast, variational SAT solving is domain agnostic, solves SAT problems expressed in VPL, returns a variational model, and uses incremental SAT solving.

Variational SAT solving is the latest in a line of work that uses the choice calculus to investigate variation as a computational phenomena. The choice calculus has been successfully applied to diverse areas of computer science, such as databases [4, 5], graphics [28], data structures [50, 67, 60, 31], type systems [14, 15, 19, 20], error messages [18, 16, 19, 17], and now satisfiability solving. Our use of choices is similar to the concept of *facets* [6] and *faceted execution* [56, 51, 7], which have been successfully applied to information-flow security and policy-agnostic programming.

The use of compiler optimization techniques in SAT solvers is not novel, for example common subexpression elimination (CSE) and variable elimination has been successfully implemented in SAT solvers[24, 53]. Other pre-processing techniques such as removing blocked clauses [36], and detecting autarkies [44] have been very successful at automatically increasing performance of the SAT or SMT solver. From this perspective, our use of choices is a pre-processing technique to speed up incremental solving over sets of related problems.

Some solvers such as z3 [23], allow users to program the heuristics used by the solver to find choose efficient solving techniques, z3 in particular calls these constructs *strategies*. Strategies are thus similar to many optimizing compiler techniques [1], only applied to the SAT domain

Chapter 8: Conclusion

SAT and SMT solvers are ubiquitous and powerful tools in computer science and software engineering. Incremental SAT and SMT provide an interface that supports solving many related problems efficiently. However, the interface could be automated and improved.

The goal of this thesis is to explore the design and architecture of a variational satisfiability solver that automates and improves on the incremental SAT interface. Through the application of the choice calculus the interface can be automated for satisfiability problems, the solver interaction can be formalized and made asynchronous, and the solver is able to directly express variation in a problem domain.

The thesis will present a complete approach to variational satisfiability and satisfiability modulo theory solving based on incremental solving. It will include a method to automatically encode a set of Boolean formulae into a variational propositional formula. A method for detecting the difficulty of solving such a variational propositional formula. A data set suitable for future research in the SAT, SPL and variation research communities. A variational satisfiability solver, an asynchronous variational satisfiable modulo theory solver and a proof of variational preservation.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986. ISBN 0201100886.

[2] 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.

[3] 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.

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

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

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

[7] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321440. doi: 10.1145/2465106.2465121. URL https://doi.org/10.1145/2465106.2465121.

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

[9] 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. ISBN 0262510952.

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

[11] 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. ISBN 1586039296.

[12] 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, 2009.

[13] 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. ISSN 1084-4309. doi: 10.1145/1698759.1698762. URL https://doi.org/10.1145/1698759.1698762.

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

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

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

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

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

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

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

[21] 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.

[22] 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. ISBN 9781450374644. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.

[23] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[24] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, SAT'05, page 61–75, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540262768. doi: 10.1007/11499107_5. URL https://doi.org/10.1007/11499107_5.

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

[26] 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. ISBN 978-3-540-24605-3.

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

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

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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. ISBN 9783642316111. doi: 10.1007/978-3-642-31612-8_12. URL https://doi.org/10.1007/978-3-642-31612-8_12.

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

[36] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for cnf formulas. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, page 357–371, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 364216241X.

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

[38] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. doi: 10.1017/S0956796897002864.

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

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

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

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

[43] 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.

[44] Mark Liffiton and Karem Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, pages 182–195, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-79719-7.

[45] 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. ISBN 3540372067. doi: 10.1007/11814948_16. URL https://doi.org/10.1007/11814948_16.

[46] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, May 1999. ISSN 0018-9340. doi: 10.1109/12.769433. URL https://doi.org/10.1109/12.769433.

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

[48] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL https://doi.org/10.1145/367177.367199.

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

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

[51] K. Micinski, D. Darais, and T. Gilray. Abstracting faceted execution. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 184–198, 2020. doi: 10.1109/CSF49147.2020.00021.

[52] 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. ISBN 978-3-319-09284-3.

[53] Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving sat encoding of constraint problems through common subexpression elimination in savile row. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, pages 330–340, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23219-5.

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

[55] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.

[56] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *CCS '18*, 2018.

[57] 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. ISBN 978-3-540-44798-6.

[58] 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. ISBN 0-8186-7597-7. URL http://dl.acm.org/citation.cfm?id=244522.244560.

[59] 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.

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

[61] 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.

[62] 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.

[63] Frank van Harmelen, Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation*. Elsevier Science, San Diego, CA, USA, 2007. ISBN 0444522115.

[64] 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.

[65] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393665. URL http://doi.acm.org/10.1145/2393596.2393665.

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

[67] 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.

[68] 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. ISBN 1581132972. doi: 10.1145/378239.379019. URL https://doi.org/10.1145/378239.379019.

APPENDICES

Appendix A: Redundancy

This appendix is inoperable.