AN ABSTRACT OF THE DISSERTATION OF

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

Title: Variational Satisfiability Solving

Abstract approved: _____

Eric Walkingshaw

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

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

Variational Satisfiability Solving

by

Jeffrey M. Young

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented September 23, 2011
Commencement June 2012

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

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

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

Jeffrey M. Young, Author

ACKNOWLEDGEMENTS

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

TABLE OF CONTENTS

Todo list

Chapter 1: Introduction

One of the most important aspects of any programming language is the ability to control complexity, especially as software written in that language grows. The burgeoning field of *variation theory* and *variational programming* [10, 15, 16, 21, 40] attempt to control complexity which is induced into a software artifact when many *similar yet distinct* kinds of the same software artifact must coexist. For example, software is often *ported* to other platforms, creating similar, yet distinct instances of that software which must be maintained. Such instances of variation are ubiquitous: Web applications are tested on multiple servers; programming languages maintain backwards compatibility and so do software libraries; databases evolve over time, locale and data; and device drivers must work with varying processors and architectures. Variation theory and variational programming have been successful in small systems, yet it has not been tested in a performance demanding practical domain. In the words of Joe Armstrong [4], "No theory is complete without proof that the ideas work in practice"; this is the central project of this thesis, to put the ideas of *variation* and *variational programming* to the test in the practical domain of satisfiability solving (SAT).

cite variational data structures and images

The major contribution of this thesis is the formalization of a *variational propositional logic (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 [7] 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 [11], SAT solvers based on conflict-driven clause learning (CDCL) [5,27,32] have been able to solve boolean formulae with millions variables quickly enough for use in real-world applications [38]. Leading to their proliferation into several fields of scientific inquiry ranging from software engineering to Bioinformatics [19,26].

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 [9,12,13,17,31,33,41] require solving a set of related SAT problems [13,31,33]. To take just one example, software product-lines (SPL) utilizes SAT solvers for a diverse range of analyses including: automated feature model analysis [6,18,36], feature model sampling [29,39], anomaly detection [3,25,28], and dead code analysis [35].

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) [31,41] 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 [14].

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 control flow operations. 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 control-flow operators. 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 control flow operations 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.

1.1.1 Contributions and Outline of the 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. In addition to providing an algorithm to generate a collapse sets of non-variational strings to a single variational string. The rest of this section outlines the thesis and expands on the contributions of each chapter:

1. *Variational propositional logic*: SAT solvers input and operate on sentences in propositional logic [8]. Variational satisfiability solvers, in order to reason about variation, must input sentences in a propositional logic that is *variational*, i.e. a many-valued logic [30] which contains variational values, called *choices* in addition to boolean values.

   The formulation of VPL is requisite and central to the high level goal of designing a variational satisfiability solver. Furthermore, VPL serves two other functions: It provides an avenue for future work through the formalization of variation in the domain of propositional logic for variational satisfiability solvers. It provides a foundation for research on variation in propositional logic outside of the considerations of satisfiability solvers.

   This work is nearly complete. The logic has been formalized and successfully used in a prototype variational solver [42]. **??** introduces VPL and describes the following contributions which are directly enabled by it:

   (a) ✓ A set of variation preserving equivalences. Similar to the well known propositional logic equivalences, such as DeMorgan's law, these equivalences allow a variational solver to refactor input possibly yielding simpler variational sentences.

   (b) An efficient algorithm for translating a set of propositional formulae into a single VPL formula. The prototype variational SAT solver used a naive algorithm, and preliminary results showed that the encoding impacts solver performance. Hence, finding a more efficient encoding algorithm is desirable. This work is yet to be done but there are two promising routes forward. First, a naive algorithm which interleaves syntactic equivalences to produce a VPL formula that is easier to solve. Second, an algorithm similar to Huffman codes [22] to translate the SAT problems into a data structure, then use heuristics to select high quality candidates to combine. With such an algorithm the end-user of the variational solver only needs to input their problem sequence rather than a VPL formula.

2. *A variational satisfiability solver*: This is the central contribution of my thesis. It is completed and is published in a peer-reviewed conference [42] paper. Preliminary results are promising but based on only two case studies from the SPL community.

   ?? discusses these results and provides an overview of the variational solving algorithm. The following contributions are based on this work:

(a) ✓ Formalization of a variational SAT solving algorithm that inputs a VPL formula and outputs a *variational model*.

(b) ✓ Formalization of variational models; that is satisfying assignments of values to variables in input formula that succinctly represent results in the context of variation.

(c) ✓ A method for determining the amount of variation in a given VPL formula.

(d) A method for determining the relative hardness of a VPL formula based on work in the random-SAT community [20]. This item is orthogonal to all other items and thus can be done in parallel.

3. *A concurrent variational SMT solver*: Contingent on item 2, *satisfiability modulo theories* extends SAT solvers such that they are able to reason about logical formulas in combination to *background theories*, such as arithmetic or arrays. Furthermore, with variation statically represented in a VPL formula, the SAT or SMT procedure can be made asynchronous leading to speedups on multi-core machines. The approach is to change the semantics of a choice; in the prototype SAT solver each choice blocks future SAT problems from being solved, by creating an asynchronous solving algorithm these future problems are unblocked and can be processed earlier.

   This item is an extension of the central contributions of the thesis. There are two extensions to the previous work to construct a variational SMT solver and one to make it asynchronous.

   First, the extensions to VPL abstract logical connectives in VPL allowing for theories which conclude to a Boolean value, such as arithmetic inequalities, and thus can be reasoned about in a SMT solver. Second, variational models are similarly extended, rather than assuming only Boolean values, the extension allows for polymorphic results through the use of SMTLIB2 compliant functions.

   Third, I extend the semantics of choices in the variational SAT solver to include atomic concurrent operations. When a choice is observed the solver state is copied and sent to a thread with instructions to compute continue the computation.

   This work is completed but unpublished. **??** expands on this item and discusses the evaluation of the prototype variational SMT solver with additional case studies. The following summarizes the expected contributions:

(a) ✓ Formalize the extension of VPL with SMT theories.

(b) ✓ Formalize the extension of variational models to express SMT results.

(c) ✓ Formalize the asynchronous variational solving algorithm

(d) A set of optimizations based on work on nanopass compilers [23] from the scheme programming language community [2]. The goal is to leverage VPL equivalence rules and other compiler optimizations, such as inlining, on SMTLIB2 programs, thus optimizing variational SMT programs. The prototype variational SMT solver is architected as a nanopass compiler and thus is able to perform optimizations as a single pass over the input formula. However, no optimizations are performed as of yet, although all requisite items for this work to begin are done.

(e) An empirical evaluation of solver performance. The empirical evaluation will reuse the datasets the prototype SAT solver was evaluated on. In addition, three new data sets will be added, two by harvesting SAT problems from work on variational lexing, parsing, and type checking [24] real world software such as Busybox [1] and the Linux kernel [37], and one by generating variational SMT problems. This dataset will be used several times in the thesis and will be made public. First, as a foundation to test the encoding strategies from item 1b. Second, to evaluation the optimizations from item 3d and third, to evaluate the performance of the single threaded and multi-threaded variational SMT solver. This work is partially complete, random generation of variational SAT and SMT problems is done, as is the Busybox dataset. The remaining work is to scale the logging solution to handle the Linux kernel.

4. *Proof of variation preservation*: A proof of variation preservation is a proof that the results of the variational solvers are sound, i.e., for any variant v, if a variational solver finds $VSat(v) = True$, then $Sat(v) = True$. Both item 2 and item 3 are verified sound via property-based testing [**?**] but the variational solving algorithm itself has not been proven sound up to the soundness of the underlying incremental solver. This work is in progress using the proof assistant Agda [34] and is expected to yield such a proof. **??** discusses this item further and lists the specific tasks left to do.

1.1.2 Significance and Potential Impact

The goal of this thesis is to explore the design and architecture of a variational satisfiability solver. The solver should allow end-users to input a set of propositional formulae and output a model that is useful *without* requiring the end-user to understand or be aware of research on variation.

This work is applied programming language theory in the domain of satisfiability solvers. Many analyses in the software product-lines community use incremental SAT solvers. By creating a variational SAT solver it is likely that such analyses would 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, by exploiting results from the programming language community.

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 possible benefits, such as parallelism, researchers in other domains may attain and thereby magnifies the impact of any results produced by the variational research community. Furthermore, the result of my thesis, a variational SAT solver, provides a new logic and tool to reason about variation itself.

For researchers in other domains, a requisite result in constructing a variational satisfiability solver is a variational compiler; which translates VPL to a solver-domain programming language. Thus, while my thesis is focused on the domain of SAT solvers, this work describes a first of its kind variational compiler whose architecture may be reused to create new variation-aware tools such as build systems or programming languages. Such compilers could directly benefit from item 3d as this item describes performance improvements that *are only possible* with an explicit and static representation of variation.

1.2 Contributions and Outline of this Thesis

Chapter 2: Background

2.1 Satisfiability Solving

Chapter 3: Variational Propositional Logic

Chapter 4: Variational Satisfiability Solving

Chapter 5: Extensions to Variational Satisfiability Solving

Chapter 6: Related Work

Chapter 7: Conclusion

Wow, that really was excellent.

7.1 Fin

i the end, my only friend, the end.

Bibliography

[1] BusyBox. `https://busybox.net/`, 2020. Accessed at October 27, 2020.

[2] Working Group 1. R7rs-small standard for the scheme programming language. `https://small.r7rs.org/`. Accessed: 2020-11-16.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg, 2007.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Ian P. Gent and Toby Walsh. The sat phase transition. In *In Proc. ECAI-94*, pages 105–109, 1994.

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

[22] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[23] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 343–350, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32, 2010.

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

[26] 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.

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

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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan; Claypool, 2016.

[35] 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.

[36] 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.

[37] Linus Torvalds. Linux Operating System. `www.kernel.org`. Accessed at December 02, 2019.

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

[39] 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.

[40] 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.

[41] 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.

[42] Jeffrey M. Young, Eric Walkingshaw, and Thomas Thüm. Variational satisfiability solving. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, New York, NY, USA, 2020. Association for Computing Machinery.

APPENDICES

Appendix A: Redundancy

This appendix is inoperable.