Algebraic Techniques in Software Verification : Challenges and Opportunities

Martin Brain
University of Oxford
martin.brain@cs.ox.ac.uk

Daniel Kroening
University of Oxford
daniel.kroening@cs.ox.ac.uk

Ryan McCleeary
University of Iowa
ryan-mccleeary@uiowa.edu

***Abstract*—One of the main application areas and driving forces behind the development of Satisfiability Modulo Theory (SMT) solvers is software verification. The requirements of software verification are somewhat different to other applications of automated reasoning, posing a number of challenges but also providing some interesting opportunities. This paper brings together and summarises the algebras and structures of interest, along with some of the problems that are characteristic of software verification. It is hoped that this will allow computer algebra researchers to assess the applicability of their techniques to this challenging, but rewarding domain.**

Software verification is the prototypical application domain for Satisfiability Modulo Theory (SMT) solvers. There are many aspects of the two research fields that show a significant degree of co-evolution. For example, the central role of theories (and the theories that are available – for example bit-vectors and arrays) can be seen as a formalisation of the domain specific decision procedures that were used in early verification systems [1]. Universal quantification is challenging for most SMT solver algorithms, leading to poor performance and thus many software verification systems avoid generating quantifiers. Likewise the importance of model generation[1] is in part driven but the utility of these models for providing execution or error traces in verification systems. The co-evolution can also be seen in the SMT-LIB benchmarks which feature many benchmark collections generated by verification tools.

This paper aims to highlight some of the requirements and 'evolutionary pressures' that software verification places on SMT solver development. It is hoped that this context will help computer algebra researchers to identify, develop and refine algorithms so that they can demonstrate impact on commercial-scale software verification problems. The topics raised are a mix between challenges that have to be overcome and opportunities in under-explored / critical areas.

I. Formula Shape

Challenge: Tolerate irrelevant formulae. The formulae generated by verification tools tend to be very different from those generated by human modelling of problems. One of the real challenges of mathematical modelling is reducing the problem to a minimal core which captures the key challenge. This is a process which requires considerable intuition and experience and it is not clear whether automating it is even possible. Consequentally, compared to directly human generated formulae, the formulae generated by verification tools are much larger and contain a significant amount of 'simple' and 'irrelevant' parts.

Opportunity: Exploit disequalities and if-then-else. Software verification problems tend to produce formulae with some Boolean structure, particularly from the use of if-then-else expression to model different paths of execution. One of the advantages of the DPLL(T)[2] algorithm is that theory solvers do not need to handle the Boolean structure of the formula. The SAT solver will provide a partial assignment to the theory literals and the theory solver must determine if they are consistent. Unlike traditional algebra problems, this includes equalities that are assigned to false and if-then-else operations. Exploiting this additional information is likely to improve performance but may require (or enable) new approaches to some problems.

Challenge: Handle inequalities. Another feature common in software verification problems that is not classically algebraic is the use of inequalities. Although there are some applications, such as equivalence checking, that are possible without inequalities, the frequency of ordering comparisons in most real software means these are a critical requirement.

II. Algebras and Structures of Interest

As well as having formula structure unlike conventional, human generated algebraic problems, software verification problems also tend to use unconventional algebras and relational structures. A minority of software verification systems use integers[2] and reals to model variables, particularly those focused on algorithm rather than program verification. However this approach seems to becoming less common as these do not capture the full behaviour of real systems (overflow, rounding, etc.) and also increase the complexity of the decision procedure.

A. Bit-Vectors

Core to the performance of SMT solvers on software verification problems is their handling of the theory of fixed width bit-vector[3]. Variables in this theory are interpreted as bit-vectors of a given size, operations are a mix of (modular)

[1]Generating a model or witness is often regard as part of a satisfiability check, particularly for Boolean satisfiability but is not formally required.

[2]Sometimes referred to as "mathematical integers" to distinguish them from the integer types in programming languages.

arithmetic (plus, multiply, divide, etc.), logical operations (and, xor, shift, etc.) and structural (extract, concatinate, etc.) with predicates for signed and unsigned comparisons. Although there have been some interesting alternatives proposed[4], [5], most solvers still use "bit-blasting"; converting the expressions to a Boolean circuit and using a SAT solver. Optimal circuits are known for some operations[6] but others such as multiplication, division and variable shifts introduce additional difficulty which can, in some cases, cause the solver to time-out for even "simple" queries. Given this fundamental limitation of current solvers and the significance of bit-vectors to application performance, better handling of these formula would be of major benefit.

Opportunity: Sub-algebras. The range of operations in the theory of bit-vectors makes it hard to find decision procedures that work well for arbitrary formulae. However for many applications, groups of variables will only ever use a handful of operations. For example, variables used as counters will only assign constants, increment and check against bounds, variables that are used as bitmaps are rarely used in multiplications and the bitwise operations form a Boolean algebra, many modern cryptographic algorithms only use add, roll and xor (ARX) and floating-point operations make heavy use of a max-plus algebra with signed comparison. Thus finding useful sub-signatures of the theory of bit-vectors, identifying their algebraic structure and then building decision procedures that exploit this structure seems like a promising research direction.

B. Floating-Point

Many software verification applications concern control systems which must use sensor data to control a real-world system (aircraft, train, industrial robots, UAVs, autonomous cars, "cyber-physical systems", etc.). Pure integer and fixed-point control systems are becoming a rarity with most control systems using floating-point. Thus there is a significant commercial drive for verification systems to handle floating-point numbers efficiently and effectively.

The so-called "standard model" converts floating-point expressions to real expressions. For example, if $\mathtt{f}$ and $\mathtt{g}$ are floating-point variables and $\oplus_{\mathtt{r}}$ denotes floating-point addition with rounding mode $\mathtt{r}$:

$$\mathtt{f} \oplus_{\mathtt{r}} \mathtt{g} \qquad \rightsquigarrow \qquad (f+g)(1+\delta) \qquad |\delta| \leqslant \epsilon$$

where ϵ is a small, format dependent constant. Although this model is simple, it has a number of limitations. An extensive set of side condition are required to correctly model actual hardware (no overflows, no subnormal numbers, etc.) and often these are the precise conditions we are trying to identify. If these are used then the standard model is an over-approximation of the behaviour of hardware, meaning that spurious SAT results can be given. Worse from an application point of view is that it is difficult to generate the bit-exact traces needed to assess, diagnose and fix real systems.

As a consequence, a theory of floating-point numbers[7] has been developed and added to the SMT-LIB standard. Its formalisation makes use of sets $\mathbb{F}_{e,s}$ of floating-point numbers with e exponent and s significand bits, plus $\mathbb{R}^*$ an extension of $\mathbb{R}$ with infinities and not-a-number. A family of order-preserving functions $\mathrm{v}_{\mathbb{F}_{e,s}} : \mathbb{F}_{e,s} \to \mathbb{R}^*$ give the real value of a floating-point number and round picks between the adjoints of v and applies one, to map back into $\mathbb{F}_{e,s}$, thus:

$$\mathtt{f} \oplus_{\mathtt{r}} \mathtt{g} \qquad \rightsquigarrow \qquad \mathrm{round}(r, \mathrm{v}(\mathtt{f}) + \mathrm{v}(\mathtt{g})).$$

This approach allows bit-exact specification of floating-point numbers with a minimal number of edge-cases and a maximal amount of algebraic structure.

Opportunity: Algebraic techniques on $\mathbb{R}^*$. $\mathrm{v}_{\mathbb{F}_{e,s}}$ has relatively little algebraic structure – floating-point addition and multiplication are famously non-associative. As not-a-number is absorbing for all operations, $\mathbb{R}^*$ is not a field but it is an additive and multiplicative commutative monoid with the associativity property [8] and so semi-ring decision procedures may be applicable.

Opportunity: Mixed real and float. If v and round can be handled then this may not only be able to handle floating-point problems but also mixed float and real problems which are of considerable commercial interest.

C. Differential Theories

Another interest driven by control systems is that of reasoning about differential equations. Most control systems are developed with respect to a model of the environment with which they are interacting. This is normally formalised as a system of differential equations. However due to limitations of the solver technology this is not directly used in the verification, instead a computational model is built. If solvers could handle differential equations of some form, it could significantly increase the ability of tools to verify these cyber-physical systems.

Opportunity: Symbolic algorithms for differential systems. dReal [9] has made some impressive progress using numerical methods for handling differential equations. Using symbolic algorithms has yet to be explored but may have great potential.

III. Beyond Satisfiability

Determining formula satisfiability is the core role of SMT solvers. However there are a number of similar and related symbolic reasoning tasks, both inside and outside the solver, that have significant practical impact and present interesting opportunities for applying algebraic techniques.

A. Expression Simplification

Expression simplification is a vital component in most SMT solvers and program verification systems. It is used for performing constant folding (`(x + 1) + 1` $\rightsquigarrow$ `x + 2`), simplifications (`x <= x` $\rightsquigarrow$ `true`) and normalisation (`!(x >= y)` $\rightsquigarrow$ `x < y`). Although the rewriting steps are generally simple, their effects can be drastic. For example, given unsigned integers `x`, `y` and `z` of more than 16 bit:

```
assert((x * y) * z == x * (y * z));
```

will cause most bit-vector decision procedures to time out but can be trivially resolved by a distributivity aware simplifier.

Opportunity: Improved simplification For such a key component of solvers, there is relatively little in the conventional published literature. There are papers that mention the number of simplification rules used [10] or discuss their importance to particular applications [11]. However there seems to be little discussion of whether these sets are complete, whether the popular architecture (stateless / context-free rewrites implemented as ad-hoc tree-walks) is best, whether they could or should be used during the solve process rather than just before. Given the similarity to rule-based differentiation and integration algorithms, there might be techniques, best practices and ideas from the computer algebra community that could revolutionise the role of expression simplification in SMT solver.

B. Fixed-Points and Approximation

Current algorithms and verification systems are often built "on top of" SMT solvers. The solver (or solvers) are the lowest level component in the algorithm and satisfiability queries are used as a way of checking inclusion or intersection between sets (represented symbolically, using equations). This architecture has proven flexible and effective. However if the reasoning system can perform more than just satisfiability queries it can be used at a higher level within verification algorithms. This section presents verification algorithms 'from the top down', highlighting the other kinds of symbolic reasoning task that arise.

1) The Verification Question: We present a simple model of software verification using the transition systems view. This is perhaps idiomatic of the model checking community, especially hardware model checking but it nicely illustrates some of the key challenges.

Let x be a set of variables and v a set of values. A map $s : x \to v$ that assigns value to variables is referred to as a *state* (in the context of transition systems) or a *valuation* (in the context of logical formulae). Let $states_{x,v}$ denote $x \to v$, the space of all such maps. Critical to *symbolic* verification is the idea that we can use a formula (over x) to represent a set of states. To make the distinction between sets represented by formulae and sets described by other means, we will use $[\![F(x)]\!]$ to denote *the set of all valuations (states) that satisfy formula* $F(x)$.

A symbolic representation of a *transition system* requires a formula for the initial states, $I(x)$ and a formula that describes the transition relation between sets $T(x, x')$. As our interest will be in the sets of states which are reachable from a given set of states, we define forwards and backwards reachability steps:

$$\begin{aligned} & FRStep : 2^{states_{x,v}} \to 2^{states_{x,v}} \\ FRStep(S) &= S \cup \{t \in 2^{states_{x,v}} | T(s,t) \wedge s \in S\} \\ & BRStep : 2^{states_{x,v}} \to 2^{states_{x,v}} \\ BRStep(S) &= S \cup \{t \in 2^{states_{x,v}} | T(t,s) \wedge s \in S\} \end{aligned}$$

Both of these are monotonic and increasing on $2^{states_{x,v}}$, a complete lattice, so by the Knaster-Tarski theorem they have fixed-points which form a complete lattice [12]. We use lfp to denote the least fixed-point of a function and define LFP to give the least fixed-point above a given set (S):

$$LFP : Mono(2^{states_{x,v}} \to 2^{states_{x,v}}) \times 2^{states_{x,v}} \to 2^{states_{x,v}}$$

$$LFP(f)(S) = lfp(\lambda X.f(X \cup S))$$

For simplicity we will consider *safety* properties; those which are false if there is an execution trace from an initial state to an unsafe state. $P(x)$ is a formula describing the set of safe states and thus:

$$\begin{aligned} FReach &= LFP(FRStep, [\![I(x)]\!]) \\ BReach &= LFP(BRStep, [\![\neg P(x)]\!]) \\ \texttt{system safe} &\Leftrightarrow FReach \subseteq [\![P(x)]\!] \\ \texttt{system safe} &\Leftrightarrow BReach \cap [\![I(x)]\!] = \emptyset \end{aligned}$$

If a formula describing $FReach$ or $BReach$ can be found, then showing system safety reduces to a single satisfiability check. However there are no guarantees that the fixed-point can be described by a formulae and even when they can, there are few algorithms that can compute them. For example, consider the following two loops, with variables in $\mathbb{N}$, one has a simple, computable fix-point, the other remains an open question:

```
while (i < n) {
  a[i] = n;
}
```

```
while (i != 1) {
  if (i % 2 == 0) i = i/2;
  else i = 3*1 + 1;
}
```

Opportunity: Finding exact fixed-points. There are many interesting questions related to exact descriptions of fixed-points; both theoretical (When does an expression over language L have a fixed-point representable in L'? Is finding it computable?) and practical (What are algorithms to compute them? Are there subsets of expressions for which fixed-point formula can be easily found?).

2) Approximate Approachs to Verification: When faced with a intractable or computationally expensive problem, a common approach in software verification is to approximate. If:

$$\begin{aligned} FReach \subset O \wedge O \subseteq [\![P(x)]\!] &\Rightarrow \texttt{system safe} \\ U \subseteq FReach \wedge \neg(U \subseteq [\![P(x)]\!] &\Rightarrow \neg\texttt{system safe} \end{aligned}$$

Thus we have reduced the problem from computing fixed-points to finding (formulae that describe) sets of states that approximate the fix-points. From the definition of $FReach$ we have necessary and sufficient conditions for over and under approximations:

$$LFP(FRStep_o, Init_o) \subseteq O$$
$$[\![I(x)]\!] \subseteq Init_o \wedge FRStep \leqslant FRStep_o \quad (1)$$

$$U \subseteq LFP(FRStep_u, Init_u)$$
$$Init_u \subseteq [\![I(x)]\!] \wedge FRStep_u \leqslant FRStep \quad (2)$$

Different approaches to verification can be seen as different ways of finding solutions to equations (1) or (2). Some of these are known as *property directed* and make of $P(x)$ so they produce an approximation that allows the safety of the system to be determined. For example, the Hoare logic system uses inductive invariants; formula $Inv(x)$ such that:

$$I(x) \Rightarrow Inv(x) \qquad Inv(x) \wedge T(x, x') \implies Inv(x').$$

This is a sufficient condition for (1) and has the advantage that it is stated purely in terms of formulae, reducing the problem to existential second-order logic. Bounded model checking aims using a sequence of solutions of (2), first $I(x)$, then

$$I(x) \vee (I(x^*) \wedge T(x^*, x)),$$

next

$$I(x) \vee (I(x^*) \wedge T(x^*, x)) \vee (I(x^{**}) \wedge T(x^{**}, x^*) \wedge T(x^*, x)),$$

and so on. These do not have fixed-points or second-order variables and thus can be solved with simple satisfiability calls. Abstract interpretation can be seen as picking $FRStep_o$ (abstract transformer) and $Init_o$ (initial abstract state) in such a way that $LFP(FRStep_o, Init_o)$ is computable in the abstract domain.

Opportunity: Finding fixed-point approximations. As with computing exact fixed-points, there are both theoretical and practical challenges. Finding new sufficient solutions to (1) and (2), especially those without fixed-points or, ideally, second-order quantification would likely yield new algorithms. A classification theorem of what solutions exist or necessary conditions would also be of great interest. Practically, algorithms that compute over or under approximations, either using one of the sufficient conditions (such as finding invariants) or directly (as abstract interpretation does) are of great interest.

3) Pre-Image, Post-Image and Merging: One approach to computing solutions to (1) and (2) is to perform stepwise approximations. This requires approximating the pre or post image of the transition relation and being able to merge two or more approximations. Pre and post image approximations can be described as finding the strongest (setwise smallest) $O(x)$ or the weakest (setwise largest) $U(x)$ that satisfies the appropriate formulae (given either Pre or $Post$):

- Forwards Over : $Pre(x) \wedge T(x, x') \Rightarrow O(x')$
- Forwards Under : $U(x') \Rightarrow Pre(x) \wedge T(x, x')$
- Backwards Over : $Post(x') \wedge T(x, x') \Rightarrow O(x)$
- Backwards Under : $U(x) \Rightarrow Post(x') \wedge T(x, x')$

Although these involve second-order quantification (we are searching for formula), they do not involve fixed-points and are significantly more practical.

Merging of approximations is needed when multiple control flow paths converge, for example after an `if` statement or after a loop. It is sufficient to take the disjunction of the two formula but this tends to lead to unacceptable growth in formula size. Thus it is useful to be able to find the smallest $O(x)$ or largest $U(x)$ such that:

$$O_1(x) \vee O_2(x) \Rightarrow O(x) \qquad U(x) \Rightarrow U_1(x) \vee U_2(x)$$

Opportunity: Step-wise Approximations. Computing abstract pre and post-images can clearly be seen in algebraic terms. Quantifier elimination, either via CAD [13] or virtual term substitution [14], can give exact results for these, is it possible to modify them to give faster under or over approximate answers? Computing a convex hull (or the dual, 'convex interior') is a way of merging, as is formula simplification. Out of all of the opportunities high-lighted this seems to be the one the is most directly accessible and closest to existing work in computer algebra.

4) Reducing Second-Order Quantification: The preceding sections have reduced computing system correctness to finding formulae with the required properties; effectively solving existential second-order logic. *Templates* are a commonly used (for example [15]) technique to reduce second-order quantification to first-order quantification. For example, using a template of $l \leqslant x \wedge x \leqslant u$, computing a (property directed) invariant can be reduced to finding (vectors of) constants l and K such that:

$$\begin{aligned} I(x) &\Rightarrow l \leqslant x \wedge x \leqslant u \\ l \leqslant x \wedge x \leqslant u \wedge T(x, x') &\Rightarrow l \leqslant x' \wedge x' \leqslant u \\ l \leqslant x \wedge x \leqslant u &\Rightarrow P(x) \end{aligned}$$

which requires a solver that can handle quantification alternation but is purely first-order logic. In abstract interpretation terms the template can be seen as characterising an abstract domain [15]; the one given in the example is an interval abstraction.

Opportunity: Template algorithms. Algorithms for working with specific templates are of great utility as evidenced by the range of abstract domains that are currently in use. Algorithms that can work with arbitrary templates or templates that are monotonic with respect to the constants have received some initial investigation but there are likely to be significant theoretical and practical gains to be made. Efficient implementations of such algorithms allow user specified templates but also allow templates to be chosen automatically, effectively refining the abstraction.

IV. Conclusion

Modern software verification techniques are heavily dependent on efficient SMT solvers. Correspondingly SMT solver development is often motivated, inspired and justified by software verification applications. It is hoped that this paper acts as a guide for computer algebra researchers to understand this synergy, and appreciate some of the places algebraic approaches could be fruitfully deployed and to get involved!

REFERENCES

[1] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, May 2005.

[2] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," ser. LNCS, R. Alur and D. A. Peled, Eds., vol. 3114. Heidelberg, Germany: Springer-Verlag, July 2004, pp. 175–188.

[3] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB standard: Version 2.5." [Online]. Available: http://smt-lib.org

[4] L. Hadarean, K. Bansal, D. Jovanović, C. Barrett, and C. Tinelli, *A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors*. Cham: Springer International Publishing, 2014, pp. 680–695.

[5] A. Zeljić, C. M. Wintersteiger, and P. Rümmer, *Deciding Bit-Vector Formulas with mcSAT*. Cham: Springer International Publishing, 2016, pp. 249–266.

[6] M. Brain, L. Hadarean, D. Kroening, and R. Martins, "Automatic generation of propagation complete sat encodings," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Jobstmann and M. K. R. Leino, Eds., vol. 9583. Heidelberg, Germany: Springer-Verlag, January 2016, pp. 536–556.

[7] M. Brain, C. Tinelli, P. Ruemmer, and T. Wahl, "An automatable formal semantics for ieee-754 floating-point arithmetic," SMT-LIB, Tech. Rep., 2014. [Online]. Available: http://smt-lib.org/papers/BTRW14.pdf

[8] ——, "An automatable formal semantics for ieee-754 floating-point arithmetic," in *IEEE Symposium on Computer Arithmetic*, 2015.

[9] S. Gao, S. Kong, and E. M. Clarke, *dReal: An SMT Solver for Nonlinear Theories over the Reals*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 208–214.

[10] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, *The MathSAT5 SMT Solver*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 93–107.

[11] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Heidelberg, Germany: Springer-Verlag, July 2007, pp. 519–531.

[12] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications." *Pacific J. Math.*, vol. 5, no. 2, pp. 285–309, 1955. [Online]. Available: http://projecteuclid.org/euclid.pjm/1103044538

[13] J. Davenport and M. England, "Recent Advances in Real Geometric Reasoning," in *Proceedings ADG 2014*, F. Botana and P. Quaresma, Eds., 2015, pp. 37–52.

[14] M. Kosta and T. Sturm, "A Generalized Framework for Virtual Substitution," http://arxiv.org/abs/1501.05826, 2015.

[15] M. Brain, S. Joshi, D. Kroening, and P. Schrammel, "Safety verification and refutation by k-invariants and k-induction ," in *Static Analysis*, ser. Lecture Notes in Computer Science, S. Blazy and T. Jensen, Eds., no. 9291. Heidelberg, Germany: Springer-Verlag, September 2015, pp. 145–161.