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 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. Consequently, 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 disequalities 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 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)
arithmetic (plus, multiply, divide, etc.), logical operations (and, xor, shift, etc.) and structural (extract, concatenate, 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 bit-wise 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 \( f \) and \( g \) are floating-point variables and \( \oplus_r \) denotes floating-point addition with rounding mode \( r \):

\[
f \oplus_r g \leadsto (f + g)(1 + \delta) \quad |\delta| \leq \varepsilon
\]

where \( \varepsilon \) 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 \( \nu_{\mathbb{F}_{e,s}} : \mathbb{F}_{e,s} \to \mathbb{R}^* \) 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:

\[
f \oplus_r g \leadsto \text{round}(v, v(f) + v(g)).
\]

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 \( \mathbb{R}^* \). \( \nu_{\mathbb{F}_{e,s}} \) 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 \((x + 1) + 1 \leadsto x + 2\), simplifications \((x <= x \leadsto true)\) and normalisation \((!(x >= y) \leadsto x < y)\). 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:

\[
\text{assert}((x * y) * z == x * (y * z));
\]

will cause most bit-vector decision procedures to time out but can be trivially resolved by a distributivity aware simplifier.
Both of these are monotonic and increasing on 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 \( S \):

\[
LFP : Mono(2^{\text{states}_{x,v}} \rightarrow 2^{\text{states}_{x,v}}) \times 2^{\text{states}_{x,v}} \rightarrow 2^{\text{states}_{x,v}}
\]

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

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:

\[
FReach = LFP(FRStep, \llbracket I(x) \rrbracket)
\]

\[
BReach = LFP(BRStep, \llbracket \neg P(x) \rrbracket)
\]

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 formula 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:

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

```plaintext
while (i != 1) {
    if (i % 2 == 0) i = i/2;
    else i = 3*i + 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 Approaches to Verification:** When faced with an intractable or computationally expensive problem, a common approach in software verification is to approximate. If:

\[
FReach \subseteq O \land O \subseteq \llbracket P(x) \rrbracket \Rightarrow \text{system safe}
\]

\[
U \subseteq FReach \land \neg(U \subseteq \llbracket P(x) \rrbracket) \Rightarrow \neg\text{system safe}
\]

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:
Merging of approximations is needed when multiple control flow paths converge, for example after an $l \leq x \leq u$ 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:

$$O_1(x) \lor O_2(x) \Rightarrow O(x) \quad U(x) \Rightarrow U_1(x) \lor U_2(x)$$

**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 highlighted 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 $l \leq x \land x \leq u$, computing a (property directed) invariant can be reduced to finding (vectors of) constants $l$ and $K$ such that:

$$I(x) \Rightarrow l \leq x \land x \leq u$$

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


