Formal Verification with SMT Solvers: Why and How

Ian Johnson

Abstract. There have been various methods devised to verify the correctness of programs that range from undecidable computational logics describing the semantics of a program to the more brute-force method of exploring the entire state space of a program to ensure certain properties are never violated.
In this paper we advocate that a compromise between the two, a decidable algorithm for exploring an abstracted state space of a program to determine if given properties are violated. This is the approach of verification with SMT solvers, and we describe the satisfiability solving algorithm along with ways in which that algorithm can interact with a theory solver. We describe the current state-of-the-art SMT solver, which relies on lazy iteration with on-line interaction and layers of increasingly complex theories instead of the more explicit, fully built model with the theory solver invoked only at the end of the satisfiability solver algorithm.

We discuss the current limitations with this approach as well as the current trends in the research community to improve performance of the solvers and expressiveness of the theories.

1. Introduction

In the world of Turing machines where asking anything in general is undecidable, we are led to ask the question, "in which domain can we ask questions decidably?" Other than the automata theorist's answer of linearly-bounded automata, in which discerning whether our program is linearly bounded is undecidable, we know there are various areas of first order logic and arithmetic that are decidable, and that we can reason about our programs in such theories. Linear arithmetic, uninterpreted functions and arrays are examples of such useful decidable theories that we can apply to dispute a program's correctness.

In this paper we will discuss a variety of topics related to formal verification and SMT solving. First we will motivate our willingness to use SMT, citing results that will be later established. Second, we will describe at a high level the evolution of approaches to SMT solving algorithms, and what the current trend in research in SMT is. Next we will give a more complete description of these approaches, and what the current state-of-the-art is in terms of parametrized SMT solvers. The solvers aside, we will describe various useful theories that have been created to reason about programs.

2. Motivation

Before going into the specifics of satisfiability solvers and certain theory solvers, we must motivate our willingness to do so. Firstly, many programs can be mechanically translated from code to formulas that an SMT solver can decide. This means no painstaking work of creating an entire model of one's machine to reason about the assembly instructions being executed in a domain where you are either (a) right or (b) wrong or not smart enough to prove you are right. Secondly, depending on the obviousness of one's error in the code, lesser theories that are decidable with far less complexity can often detect errors without resorting to more computationally intensive and more complete theories. Finally, although satisfiability is the classic NP-complete problem, and modulo

theories possibly even more complex, the gap between engineered programs and theoretically possible programs is rather large, and the technology that currently exists for SMT is overwhelmingly fast. When a formal methods expert is asked their opinion about P versus NP, the common answer is, "I eat NP-completeness for lunch." [1].

A reasonable question an astute reader might ask is, "what is meant by 'many programs', and what application domains have already adopted the use of SMT solvers in industry?" In the first case, the answer is quite a lot. There exist theories that reason about array manipulation, linear arithmetic, and rich data structures that can be modeled by linked lists [19]. However for reasoning about non-linear arithmetic, one's best bet is still a mathematician. A great variety of applications can be modeled only with linear transformations, so this is still a powerful theory to have in one's toolbox. As for application domains, there already exist plenty in the propositional sense of satisfiability, so the added layer of abstraction with SMT could (and has [17]) improve performance in the realms of circuit design [20], artificial intelligence [10] and scheduling [16].

3. Lunch

Despite the formal methods expert's comments on the ease of solving NP-complete problems, the current technology that exists is built upon 40 years of theory and empirical studies. Most of today's solvers are based on the Davis-Putnam procedure that was created in 1960 [13] and then revised for performance in 1962 [12]. For historical reasons, this is now known as the DPLL (Davis-Putnam-Logemann-Loveland) procedure. This family of satisfiability solvers leverages the separation of clauses in the conjunctive normal form (CNF) of propositional formulas.

A propositional formula is composed of literals (Boolean variables or their complements) joined by the binary operators OR and AND with parentheses to disambiguate; subterms of the formula within parentheses may also be negated. The formula is said to be *satisfiable* if there exists an assignment of truth values to the used Boolean variables in order for the entire formula to be true. If there does not exist such an assignment, it is called *unsatisfiable*. Further, a formula is *valid* if its complement is unsatisfiable. Conjunctive normal form is a subset of all possible propositional formulas, but any propositional formula can be transformed in polynomial time to a formula in CNF that is satisfiable if and only if its counterpart is satisfiable. CNF has the form of many clauses AND-ed together (called a conjunction of clauses). A clause is simply a set of literals OR-ed together (called a *disjunction* of literals). This means that if one of the clauses is unsatisfiable, then the search may terminate.

The naïve approach to writing a satisfiability solver would be to systematically try every truth

[1] Quoted from Byron Cook at the talk "Terminator - Proving Good Things Will Eventually Happen" and private conversation with Pete Manolios

assignment of the variables. If there are n variables, then this generally means there must be 2^n decisions on variable truth assignments. The novel idea put forth by Davis and Putnam was that not every literal in a formula must be decided, but can be discerned from the current partial assignment of truths. Let us develop a notation and vocabulary to describe the workings of the DPLL procedure. Let P be a fixed finite set of propositional symbols (our *variables*). If $p \in P$ then p is an *atom* and p and $\neg p$ are *literals* of P. If a literal l is p, then $\neg l$ means $\neg p$, and if l is $\neg p$ then $\neg l$ means p (law of the excluded middle). A partial truth assignment M (for *model*) is a set of literals such that for all $p \in P$, $\{p, \neg p\} \subsetneq M$. This renders our assignment consistent, for a variable cannot be both true and false. A literal l is true in M if $l \in M$, false if $\neg l \in M$, and undefined otherwise. To denote a formula F satisfied by M, we write $M \models F$, and to talk about the truth of F given the partial assignment M, we write $M \parallel F$.

The DPLL procedure depends on the idea of a *unit*, meaning a literal l and a clause $C \vee l$ such that l is undefined in M and $M \models \neg C$. Since l is undefined, we can give its corresponding atom a truth value such that $C \vee l$ becomes true. Because of this idea, satisfiability can be determined without explicitly guessing the values of each atom. Another idea such as this, yet not as far-reaching, is finding undefined literals l such that l is in some clause and $\neg l$ is in no clause and adding l to M. This is not as far-reaching since the search is expensive and such literals do not manifest themselves during the decision process. Classically, this step was run during the procedure, but in modern solvers, this step is performed in preprocessing if at all.

Where advanced heuristics and most of the engineering behind SAT solvers resides is in the choice stage of the algorithm. Formulas tend to have a certain discernible structure to them, and thus the choice of which literal to guess the value of should not be completely unguided. Before discussing these techniques, let us first discuss decision, finding conflicts, and resolving conflicts. The notation for a decision literal is l^d, and the solver implementation will have some ability to record which literals are *decision literals*. A decision only occurs when there are no more units, and l or $\neg l$ is in some clause and is undefined. Because of the disjointness of clauses in CNF form, we do not have to decide all other literals before determining if our last choice created a conflict. A *conflicting clause* is a clause C such that $M \cup \{l^d\} \cup N \models \neg C$ where N contains no decision literals (i.e. determined by unit propagation). When a conflicting clause is encountered, then we must *backtrack* the previous decision literal and negate its value. The literal will no longer be annotated as a decision literal since it now has no choice but to be the negation.

An interesting part about modern solvers is the fact that they do not strictly follow the previously mentioned backtrack rule. Instead, there are certain machine learning techniques that are used to find more *relevant* literals higher up the decision stack that are more likely to yield a solution

once negating. This strategy is calling *back-jumping*. Machine learning is such a broad area and its name-dropping here could be interpreted just as easily as "magic", but the specific strategy of *conflict-driven back-jumping* we now introduce is well-founded in logical reasoning.

The goal of conflict-driven back-jumping systems is to generate clauses that are logical consequences of $M \parallel F$, such that if those clauses were present in F at the time of a previous decision, that decision literal could have actually been a unit. The formal rules for this are if we have $T = M \cup \{l^d\} \cup N$ and some clause C such that $T \models \neg C$ and another clause $D \vee k$ (k an undefined literal in M) such that $F, C \models D \vee k$, $M \models \neg D$, k or $\neg k$ occurs in F or in T. For such a scenario, we call C a *conflicting clause*, and D a *back-jump clause*. To find such a D in a reasonable amount of time, we use a form of machine learning which simply saves a clause it deems relevant by certain criteria. We use the criteria that clause C will be remembered if each atom of C occurs in F or in M and $F \models C$. In order to not overload the system by saving too many "useless" clauses, we can drop a clause (meaning $M \parallel F, C \Rightarrow M \parallel F$) without adverse affect on correctness of the algorithm, since these clauses are only saved to help improve performance (proof in [18]).

A question one might ask is, "when can we say a clause is 'useless'?". There are many heuristics for this that may be used, but two popular strategies are to track relevance as defined in [1], or find a clause's activity level (defined in [8]) has dropped below a given threshold. The balancing game of finding good heuristics for low overhead is a difficult one, however, so the use of learning anywhere the rule allows it is often detrimental to performance [18].

Now that we have a system of back-jumping, there seems to be little use to remember the order in which literals were decided, but in fact, a common approach to back-jumping is to create a *conflict graph*. By remembering the order of decision and the results of unit propagation, we can work backwards from the conflicting clause to build a graph of literal dependence to a decision level. With such a graph we split into three parts the nodes with no incoming edges, nodes with no outgoing edges, and the rest. The first two can divvy up any literals from the last group, since across any such cut, we see that the conflicting clause cannot be satisfied. From this, we can take non-decision literals with no incoming edges and a top level middle literals and construct many conflicting clauses to help back-jump. *The key restriction is that only one of these literals may be in the current decision level.* This literal is called a *unique implication point* (UIP) of the conflict graph. Formally, letting D be the set of all literals of C that have become false at the current decision level, a UIP is any literal that is in all paths from the current decision literal to the negation of a literal in D. The negation of a UIP acts as k in the back-jump rule. Without constructing the entire graph, if we work backwards from the conflicting clause and maintain the first level pre-image of literals not yet explored (called a *frontier*), we can find the first UIP and use that to back-jump. This strategy is used more commonly than the very broad rule that we

introduced above to learn clauses since it pinpoints conflicts more succinctly.

So there you have it: the modern approach to SAT solving. Indeed there is quite a lot of theory and experimentation that went into these methods to make them extremely fast, but because they are run by clever heuristics and not theoretically optimal procedures, SAT remains NP-complete. But so what? We are here to formally verify the correctness of programs and not to argue about P vs NP. These solvers are fast for the engineered problems that formal verifiers are designed to tackle, so we exploit that. Other optimizations exist to improve performance, such as the two-watched literal technique for finding unit literals, and random restarts to save the solver from probably futile further decisions. We will not discuss these methods, but rather refer you to [21]. Next we will discuss how to use these SAT-solving techniques to determine satisfiability of first order formulas modulo theories.

4. Theories, Strategies, Dirty Knees

As discussed in the introduction, SMT solvers decide satisfiability of functions modulo *theories*. What is a theory in this framework? Big Bang Theory, Freudian Theory, or Conspiracy Theory? No, a theory is a set of axioms and rules of inference in which first order logic predicates are interpreted. Our framework changes from the propositional case covered by the DPLL procedure above by changing the propositional atoms to first order sentences, i.e. first order logic formulas with no free variables. Formally, a theory T is the set of axioms and all deducible formulas from the rules of inference (all true statements). A formula F is T-satisfiable if $F \wedge T$ is is satisfiable in the first order sense. If not, F is T-inconsistent, or T-unsatisfiable.

Instead of the binary true or false valuations for atoms given by $l \in M$ or $\neg l \in M$, we have our partial assignment M be a set of first order sentences. With these sentences as our *literals*, if $M \models F$ as in the propositional case, then M *is a* T*-model of* F. A logical consequence in this framework is $F \models_T G$ if and only if $F \wedge \neg G$ is T-inconsistent, where F and G are formulas.

Since SAT solvers deal with CNF formulas, we will only think about conjunctions of sentences. A **decision** procedure that determines if the conjunction of sentences is T-satisfiable will be referred to as a T-solver. We now will discuss the use of T-solvers and how to connect them with DPLL to solve SMT problems.

The strategies that we discuss for solving SMT problems begin from easy and slow (slow meaning not scalable), to complex and fast (fast meaning scalable to an extent and outperforming much of its competition). This seems to always be the trend in any problem, and here is no exception. The complexity of the later introduced strategies should not be feared, however, since they are still quite comprehensible and elegant. The first family of strategies to solving SMT problems is known as *eager* techniques. One of such is to simply transform the input formula to a satisfiability-preserving

propositional CNF formula to be checked by the above DPLL procedure. This is at the cost of exponential blow-up in the size of the formula, but is a very simple approach that can use the best SAT solvers off the shelf with no modification.

The second family is that of *lazy* techniques. The old saying (well, not that old) that a good programmer is a lazy one rings true once again, as the following strategies are very much more flexible and more easily optimized that a blast translation to a propositional formula. In this approach, all atoms are treated as propositional until a satisfying assignment is found. Without regarding the background theory, we can see if at a higher level our formula is unsatisfiable, leveraging the powerful SAT solver technology of today. Once an assignment is given, the work is handed off to the T-solver as a large conjunction of literals. If the solver returns that the model is T-consistent, then the formula is satisfiable. Otherwise, the solver builds a ground clause of a conservative amount of literals that rendered the conjunction inconsistent and adds that lemma to the original formula. This continues until the SAT solver returns unsatisfiable, or the T-solver returns T-consistent. Of course, this method can be optimized by closer communication between T-solvers and the SAT solver, with techniques of which we will take a cursory overview.

5. How to be Lazy

The four (not necessarily mutually exclusive) main categories of lazy techniques that require intercommunication with SAT solvers are that of incremental solvers, on-line solvers, theory propagation and exhaustive theory propagation. The first of these techniques is based on a simple idea that the SAT solver need not construct an entire satisfying model for the T-solver to return T-inconsistent, in effect possibly saving a large amount of unnecessary work. When to send the current model to the T-solver is up to the user's tastes. It can be invoked at every decision, detecting inconsistencies as early as they are formed, at regular intervals, or some more sophisticated load-monitoring mechanism. A general rule for incremental solvers is to allow for incremental additions, meaning that adding one literal should be faster than reprocessing everything up to the current point plus the new literal. This is a rather obvious criterion, but with a large amount of cross dependence between literals, the details of how to achieve such a criterion can be tricky. The possibility of such a criterion also comes into question. It is known that incremental solvers for difference logic can meet this demand [18], but not necessarily for more complicated theories.

Next on the list is that of on-line solvers, which build off the fundamental early pruning paradigm of the incremental solver. For the above strategy, one might ask that once a T-inconsistency is found, can we use knowledge of this inconsistency to back-jump rather than restarting the search with an augmented formula? Indeed, we find a conservative set of literals that rendered the assignment inconsistent and use it as a conflicting clause to back-jump to an earlier stage.

The concept of theory propagation is to guide the assignment process of the SAT solver, rather

than take the solver's output and simply say yes or no. Shortly stated, theory propagation is a stage in the DPLL algorithm that takes the current partial assignment M, finds a literal l such that $M \models_T l$, and returns the new model $M \cup \{l\}$. In order for this literal to be useful, we require that l or $\neg l$ is in the formula. The way in which this literal is efficiently constructed will not be discussed here, but [11, 18] cover the topic in more detail.

The last strategy of exhaustive theory propagation is to eagerly perform all possible theory propagations before carrying on with the DPLL algorithm. An interesting consequence of doing this is that there is no duplication of learned theory lemmas (as opposed to some when non-exhaustive theory propagation is used). This is because any literal found by the DPLL unit propagation rule due to a learned theory lemma will in fact be caught by exhaustive theory propagation. A simple but important note to make here is that the theory in must have terminating rules of inference for exhaustive theory propagation to terminate, even though there are finitely many goals $l \in F$, since to conclude with l or $\neg l$ is to go through some number of inference rules.

6. Decidable Theory - Difference Arithmetic

The theory of difference arithmetic (aka difference logic or separation logic) is actually a subtheory of the more computationally intensive theory of linear arithmetic (even though still NP-hard) in which each inequality is restricted to the form $x - y \leq c$ for variables x, y and constant $c \in \mathbb{Z}$. The same tricks from linear program constraint specifications apply here to force equality and strict inequality ($x = y$ has constraints $x - y \leq 0$ and $y - x \leq 0$, while $x - y < c$ has constraints $x - y \leq c - 1$ and $y - x \leq -c - 1$). The key insight into deciding satisfiability of a system in difference arithmetic is to see its link to weighted directed graphs, so we can employ the Bellman-Ford algorithm incrementally to find integer assignments to the variables. Essentially, variables and each inequality are treated as vertices. The edges are from the y to the inequality in which it is used, with weight c. Sans the proof and details of finding negative-weight cycles, given in [4], the system is unsatisfiable if there is a cycle of edges such that $x_{i-1} - x_i \leq c_i$ for $1 \leq i \leq n+1$ (n the number of constraints, $x_0 = x_{n+1} = x$) where $\sum_{i=1}^{n} c_n < 0$, since there is no assignment to x or other variables in the cycle that satisfy the constraints. If the system is satisfiable, the algorithm described in [11] follows a process of starting with an arbitrary assignment and performing a series of relaxations until a negative weight cycle is found (unsatisfiable) or all edges have been successfully added to an initially empty edge map (satisfiable).

Since the scope of this paper is not to give all the specific details of these theories, but rather why they are useful, we give a few examples of where this theory has been applied successfully. There are many areas of programs that have simple difference arithmetic to set bounds of a loop or to calculate an array index. Questions we can ask of code that falls into the realm of difference arithmetic are typically of the form, "with these initial constraints, will the arithmetic in this rou-

tine cause an out of bounds exception?" or "are the properties of variable x after this manipulation what I expect?". The trick usually played to get "preconditions" such as $x \geq 10$ is to have a dummy zero variable x_0, and have the constraint $x - x_0 \geq 10$. Less obvious applications of this theory are presented in [5], in which Cotton describes a translation of the classic Job Shop Scheduling problem into a difference logic problem, and also translations of bounded model checking of timed automata and circuit timing analysis to an equisatisfiable difference logic problem. Application of difference logic to these last two is the most significant, since these areas are used pervasively in the hardware industry.

7. Decidable Theory - E-Satisfiability

Also a theory of many names, E-Satisfiability also is known as EUF or the logic of uninterpreted functions with equality. This is commonly the first theory tried in a hierarchy of increasingly complex theories, as determining satisfiability of a conjunction of predicates is $\mathcal{O}(n\ log\ n)$ complexity. This theory has a large domain as well, so it is a nice catch-all for beginning the SMT search. Since the meanings of functions are not investigated, EUF is limited in its power, but the inconsistencies it does catch will not have further cycles wasted on them. An example of an inconsistency that could be caught is

$$(f(f(a)) \neq b \vee f(f(f(b))) \neq b) \wedge f(a) = a \wedge a = b$$

Rather than simply equality, this theory can be altered for any equivalence relation, since we can use an equivalence check instead of an actual equality check in the algorithm. The algorithm presented in [14] is for a superset of EUF that allows for functions with integer inputs to integer offsets on their output and inputs (useful for hardware verification without having to resort to difference logic), but the outline given here will not allude to this extension to the logic. The T-solver proposed in [9] makes use of a special purpose congruence closure generation algorithm, which will be first explained. Initially, in order to keep the algorithm simple, functions are translated into a simpler form through a method called "Curryfying", in which functions with multiple inputs and functions as input are changed into a formula of the "apply" function, $\cdot$, and then flattened to depth ≤ 1 operations with a linear expansion of terms. An example of what is meant by this is for the example function $f(a, g(b), b) = b$, Curryfying gives $\cdot(\cdot(\cdot(f, a), \cdot(g, b)), b) = b$. Then we abstract to flatten, making the following assignments, $c := \cdot(f, a)$, $d := \cdot(g, b)$, $e := \cdot(c, d)$, and finally for the equality of the function with b, $b = \cdot(e, b)$. By having this form, we can guarantee to a congruence closure procedure that it will only work with the function $\cdot$ applied to two constants.

The congruence closure problem is in general, given two n-ary functions, f, g and an equivalence relation $\equiv$, is $f(s_1, ..., s_n) \equiv g(t_1, ..., t_n)$? A general algorithm is given in [11], but we only discuss the simplified version for the translated functions given in [14]. The problem restated for this domain is, given a set of equations E of the form $a = b$ or $\cdot(a, b) = c$, and no term $\cdot(a, b)$ is assigned to

$$\left.\begin{array}{rl} f(a) & = g(b) \\ g(c) & = h(f(c), g(a)) \\ b & = c \\ f(c) & = g(a) \\ h(d,d) & = g(b) \\ g(a) & = d \end{array}\right\} \Rightarrow \left[\begin{array}{rl} \cdot(f,a) & = e_1 \\ \cdot(g,b) & = e_2 \\ \cdot(g,c) & = e_3 \\ \cdot(f,c) & = e_4 \\ \cdot(h,e_4) & = e_5 \\ \cdot(g,a) & = e_6 \\ \cdot(e_5,e_6) & = e_7 \\ \cdot(h,d) & = e_8 \\ \cdot(e_8,d) & = e_9 \end{array}\right] + \left[\begin{array}{rl} e_1 & = e_2 \\ e_3 & = e_7 \\ b & = c \\ e_4 & = e_6 \\ e_9 & = e_2 \\ e_6 & = d \end{array}\right]$$

Figure 1: Input equivalence set before and after Curryfying

more than one constant, what is the list of congruence classes? In order for congruence classes to be resolved over many variables, there must be a *representative* of all of the class, so we will refer to the representative of a variable a to be *the* congruence class in which it belongs, but the representative is ad-hoc chosen amongst the given constant names. For brevity, we shall call "the representative of x", x' or $rep(x)$.

All of E starts off in a *pending* structure, but one at a time, an equation $a = b$ is withdrawn to change all elements x such that $x' = a'$ to have $x' = b'$. Then each equation of the form $\cdot(c, d) = e$ where $c' = a'$ or $d' = a'$ (this check is given from a *use list*), we must check if $rep(\cdot(c', d'))$ itself is equal to e'. If $\cdot(c', d')$ is a known constant f and $f' \neq e'$ then we add $e' = f'$ back to pending. The reason for this is that not every congruence will be checked before this, and though e and f might be congruent, we do not know yet. After this check, we set $\cdot(c', d')$ to be e', and add $\cdot(c, d) = e$ to the use list of b'. If not, we have detected an inconsistency. To run through a more complex example, verify with this algorithm that indeed the following set in figure 1 gives the afterwards given congruence classes in figure 2. The representative constants are given in bold.

In addition to this algorithm, the same authors created in [15] an extension to give close to minimal *explanations* for why two predicates are or are not congruent in order for small conflicting clauses to be generated for better back-jumping. This congruence closure algorithm is used to generate for each input literal a list of equalities and a list of disequalities. If at the end we have the ability to query in constant time if an equation of the form $a = b$ is true, where a and b are literals in the input set.

8. Decidable Theory - Bit Vectors

The theory of bit vectors is a very prevalent theory used in the verification of hardware. The reason for this is because the popularity of SAT-based bounded model checking gave rise to the desire to parametrize the width of words instead of always having to expand out equations down to the individual wires, expressed purely in propositional logic. Translating a circuit into this

$\{\mathbf{a}\}$ $\{\mathbf{c} = b\}$ $\{\mathbf{f}\}$ $\{\mathbf{g}\}$ $\{\mathbf{h}\}$
$\{\mathbf{e_6} = e_4 = d = \cdot(g, a) = \cdot(f, c)\}$
$\{\mathbf{e_7} = e_1 = e_2 = e_3 = e_9 = \cdot(f, a) = \cdot(g, b) = \cdot(g, c) = \cdot(e_8, d) = \cdot(e_5, e_6)\}$
$\{\mathbf{e_8} = e_5 = \cdot(h, e_4) = \cdot(h, d)\}$

Figure 2: The output congruence classes of the input equivalences

propositional form is typically called *bit blasting*. The theory of bit vectors abstracts these multiple wires into a single entity, therefore simplifying what may have caused exponential blow-up, such as assignment, concatenation, named selection (for a wire $[0, 2^n - 1]$, only select $[i : j]$), arithmetic or per-wire bitwise operators. Having bit vectors with only assignment supported is considered the base bit vector theory, but it can be augmented with the previous enumeration of operations. Deciding T-satisfiability for assignment ($\mathcal{BV}(\varepsilon)$), concatenation and named selection ($\mathcal{BV}(C)$) is known to be in P [6], but adding arithmetic ($\mathcal{BV}(CA)$) or bitwise operators ($\mathcal{BV}(CB)$) or both ($\mathcal{BV}(CAB)$) makes the theory NP-hard [3].

This paper will discuss the more interesting theory $\mathcal{BV}(CAB)$. In this theory, the ground terms are constants, variables, or the application of an operator to a term. The operators are as follows: assignment $=$, named selection $[i : j]$, concatenation $::$, arithmetic $\{+, -, *, <\}$ where $*$ is multiplication by a scalar, and bitwise operators $\{\mathbf{AND}, \mathbf{OR}, \mathbf{NOT}\}$. Since $=$ and $<$ are relations, we call the operation of either on two terms an atom. The solver proposed in [17] (which also categorizes previously tried strategies) determines T-satisfiability by using a hierarchical approach with three levels of increasing complexity and expressiveness.

After some preprocessing to the register transfer level (RTL) circuit specification to make formulas a bit more[2] canonical for faster solving, the theory solver enters the first tier of solving using EUF, since it is a generally good first try. The second layer is incomplete with respect to which formulas it can decide, but at a lower complexity than the final layer's solver. Since the second layer is the most in-depth part of the approach in [17], we give a brief description of layer 3 before continuing. The solver in layer 3 performs a reduction of the RTL circuit specifications to integer linear programming constraints (a known NP-hard problem) with details of the reduction explained in [2].

In layer 2, some more simple transformations not performed by the preprocessor are executed. The first to eliminate concatenations whenever possible, so the easiest case is a perfect match such as $t_1^m :: t_2^n = t_3^m :: t_4^n$ (superscripts denote word width) which is split into a conjunction of $t_1 = t_3$ and $t_2 = t_4$. When there is not a perfect match, the concatenations are still simplified as much as

[2]Bit vector canonization is a known NP-hard problem, so an effective and efficient polynomial-time transformation to "cleaner" formulas is used.

possible. A statement of the form $t_1^m :: t_2^n = t_3^{m+n}$ is rewritten to the conjunction of $t_1^m = t_3[m+n : n]$ and $t_2^n = t_3[n-1 : 0]$. These selections are treated as new terms in themselves and have a subset of the preprocessor's transformations performed on them. The real meat is in the deduction rules that allow use to decide satisfiability. In the list below, terms on the top of the line are what are required for the rule to fire and produce the term on the bottom, and c represents a constant.

$$\frac{\mathbf{t_1 = t_2} \quad \mathbf{t_2 = t_3}}{\mathbf{t_1 = t_3}} Tr1 \qquad \frac{\mathbf{t_1 < t_2} \quad \mathbf{t_2 < t_3}}{\mathbf{t_1 < t_3}} Tr2 \qquad \frac{\mathbf{c < t_1} \quad \mathbf{t_1 < t_2}}{\mathbf{c + 1 < t_2}} Tr3 \qquad \frac{A \quad \neg A}{\bot} Exc$$

The first two transitivity rules are rather straight forward, but we keep in mind that the third depends on the fact that $\mathbf{t_1}, \mathbf{t_2}, \mathbf{c}$ are integers. The final rule is a clear case of contradiction. In order to use these rules, the terms are rewritten for the DPLL algorithm to attempt to find a satisfying assignment. If it cannot, these rules are easily backtracked with concise conflict clauses.

We see that from this theory, the ability to do arithmetic at the integer level instead of the gate level is a great gain, since the blow-up from the bit blasting is enormous and also does not have a lot of exploitable structure in normal SAT solvers [17]. The layered approach is also a nice bit of alleviation to the state explosion problem, since the solver tries to stay as abstract as possible, and therefore does not explore a great multitude of case splits.

9. Decidable Theory - Interpreted Sets and Bounded Quantification

Up until now you may have noticed that most of the theories presented have been either low in expressibility, or only mostly applicable to hardware problems. The theory of interpreted sets and bounded quantification is an NP-complete theory that was constructed to reason about heap-manipulating programs. In other words, this theory is explicitly meant for software verification. The logic is much deeper than the previous theories, and the transformation of a program to its domain is much more complicated, so an astute reader might correctly guess that software verification is an inherently more difficult problem to tackle. This is indeed true, since hardware is almost by definition finite state and neatly organized with repeating structure, but software is a different story, since its shape depends on a vastly larger set of variables, and in general handles an enormous amount more different problems than hardware ventures.

The logic has (understandably) many restrictions on the formulas it can verify, the most significant of which is the requirement of code without loops or procedure calls, however non-deterministic statement execution is supported. This does not eliminate programs with loops and procedure calls completely, however, since the introduction of programmer-supplied (or static analysis algorithm-generated) loop invariants can give the solver sufficient information to prove a structural induction problem. Loops and procedure calls can also be unsoundly unrolled a bounded number of times in order to try to find shallow bugs. A program is specified to be a chain of statements either executed

sequentially or non-deterministically, with a statement defined in BNF as

$$T \in Statement ::= Assert(\varphi) \mid Assume(\varphi) \mid x := new \mid free(x) \mid$$
$$x := t \mid f(x) := y \mid T_1; T_2 \mid T_1 \square T_2$$

where ; denotes sequential execution and $\square$ denotes non-deterministic execution. The formulas φ are defined by the grammar

$$\begin{array}{lll}
c & \in Integer & \\
x & \in Variable & \\
f & \in Function & \\
\varphi & \in Formula & ::= \alpha \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \\
\alpha & \in \forall Formula & ::= \gamma \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \wedge \alpha_2 \mid \forall x \in S.\alpha \\
\gamma & \in GFormula & ::= t_1 = t_2 \mid t_1 < t_2 \mid t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \mid \neg\gamma \\
t & \in Term & ::= c \mid x \mid t_1 - t_2 \mid t_1 + t_2 \mid f(t) \mid ite(t = t', t_1, t_2) \\
S & \in Set & ::= g^{-1}(t) \mid Btwn(f, t_1, t_2)
\end{array}$$

where $ite(t = t', t_1, t_2)$ means, "if $t = t'$ is true, then t_1, else t_2" (if-then-else). The predicate $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ is used to mean sort reachability from term t_1 to term t_3 with term t_2 as an intermediate. Sets and the ternary reachability predicate $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$ are closely related to variables and their *types*. Each program has a finite, partially-ordered set $\mathcal{D}$ of sorts (the partial order comes into play for a technical reason not explained in this paper). For instance $Integer \in \mathcal{D}$. Each variable has a type $D \in \mathcal{D}$. Uninterpreted functions may not have the same type for their output as their input, so an uninterpreted function f has type $D \to E$ for types $D, E \in \mathcal{D}$.

Along with the abstract notion of a type is the need for the model of one, say for a type $D \in \mathcal{D}$, its model is written M_D. A model of a type is simply the set of all objects that are of that type, so $M_{Integer} = \mathbb{Z}$. The interpretation of a variable x is $M_x \in M_D$ for each x of type D. The interpretation of a function $f : D \to E$ is similar: $M_f : M_D \to M_E$. The $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ predicate is used to define reachability of one term to another through the iteration of a function $f : D \to D$, for $D \in \mathcal{D}$. Speaking in terms of a model M, $M \models t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ if and only if there are distinct $u_0, ..., u_n \in M_D$ such that $M_{f(u_i)} = u_{i+1}$ for all $i \in [0, n)$ and $u_0 = t_1$, $u_n = t_3$ and there is some $i \in [0, n]$ where $u_i = t_2$.

For the set term $g^{-1}(t)$, we get the pre-image of a function $g : D \to E$ for $D, E \in \mathcal{D}$ and $D \neq E$. In terms of models, $M \models x \in g^{-1}(t)$ if and only if $M_{g(x)} = M_t$. The second set term is for a function $f : D \to D$, and $M \models x \in Btwn(f, t_1, t_2)$ if and only if $M \models t_1 \xrightarrow{f} x \xrightarrow{f} t_2$. Due to the boundedness of quantification in this logic, quantifiers cannot be alternated in formulas.

This seems so far to be a big jumble of definitions for a construct that might be useful for

expressing certain properties, so a couple examples of formulas in this logic are given here.

Sorted List: (`hd` points to the head of a null-terminated list)

$$\forall \texttt{u} \in \texttt{Btwn(f,f(hd),null)} \setminus \{\texttt{null}\}.$$
$$\forall \texttt{v} \in \texttt{Btwn(f,u,null)} \setminus \{\texttt{null}\}.\texttt{data(u)} \leq \texttt{data(v)}$$

To make sense of this statement, we break it down term for term. We start with an α term of the form $\forall x \in S.\alpha$, where x is `u` and S is `Btwn(f,f(hd),null)` $\setminus\{$`null`$\}$. This is the set of all nodes in the list from the head to `null`, removing the `null` element. The . after this term gives us that there is an α term to follow, specifically again of the form $\forall x \in S.\alpha$. The x here is v, the S is `Btwn(f,u,null)` $\setminus\{$`null`$\}$, and the α, `data(u)`$\leq$`data(v)`. This S is the set of nodes in the list from `u` forward to `null`, and then removing the `null` element. The final α is a γ term, with $t_1 =$`data(u)` and $t_2 =$`data(v)`, which are terms of the form $f(t)$. Here f is `data`, which points a list node to its payload data. In all, this means that walking forward in this list, each element is less than or equal to all successive elements, which is a definition of sortedness.

Non-aliasing Pointers of the Same Field: (`u` an object with field `f` [e.g. `u` a list node and `f`=`data`])

$$\texttt{f}^{-1}\texttt{(f(u))} = \{\texttt{u}\}$$

Simply, the set of all objects that point to `f(u)` through `f` is singly `u` itself. This can be extended to have non-aliasing over multiple fields, however if the data worked with is only locally important, this statement is typically sufficient.

Now that a general understanding for the expressiveness of this logic, we discuss the idea behind the logic's decision procedure. Using Dijkstra's weakest precondition calculus with a couple tweaks, we show that a program T does not meet its specification if and only if the negation of the weakest precondition of T only assuming *true* is satisfiable. In other words, if the weakest statement we need for T's correctness is valid, then T is correct. The procedure for deciding this satisfiability is detailed in [19].

10. Combining Theories

In the details of the previously explored logic's decision procedure, there is an expressed need to combine the theories of uninterpreted functions and of linear arithmetic. In the field of SMT solving, the problem of combining theories is in fact a very large problem because no single theory is "one size fits all" and thus theories must be combined, though many devised methods of doing this have been costly [7]. Before going into details of theory combination methods, there are a few terms that are going to be used many times in this discussion that must first be defined. A *signature* Σ is a set of function and predicate symbols that under an interpretation would be assigned to actual functions and predicates (e.g. $\{=, +, -, *\}$). Since equality is assumed to be in every theory's signature, when

signatures Σ_1 and Σ_2 are said to be disjoint we mean $\Sigma_1 \cap \Sigma_2 = \{=\}$. A theory is *stably infinite* if every satisfiable quantifier free formula is satisfiable in an infinite model (an example of a theory that is not stably infinite is that of bit vectors[3]). Lastly a theory T is *convex* if and only if for all finite sets Γ of literals and for all non-empty disjunctions $\bigvee_{i \in I} u_i = v_i$, $\Gamma \models_T \bigvee_{i \in I} u_i = v_i$ if and only if there exists an $i \in I$ such that $\Gamma \models u_i = v_i$ (an example of a theory that is not convex is that of linear integer arithmetic because of the set $\{0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1, 0 \leq x_3 \leq 1\}$).

The task of combining theories is most easily done when the two theories have disjoint signatures, are stably infinite and convex, though technically the weakest condition needed to combine theories T_1 and T_2 is that if Σ_1 and Σ_2 are their respective signatures, then the signature $\Sigma = \Sigma_1 \cup \Sigma_2$ of theory $T_1 + T_2$ must have a projection to Σ_i of a T_i model [11].

The first technique we examine to compose T-solvers is the classic Nelson-Oppen procedure, which *purifies* formulas into Σ_1 and Σ_2 terms by propagating equalities of shared variables between the two stably infinite theories with disjoint signatures. More technically speaking, for Γ a set of literals over $\Sigma_1 \cup \Sigma_2$ and $\mathcal{V}(\Gamma_i)$ the set of variables in Γ_i, we use the Nelson-Oppen procedure to transform Γ into $\Gamma_1 \wedge \Gamma_2$ such that the symbols from Γ_i are in $\Sigma_i \cup \alpha$ where $\alpha = \{\mathcal{V}(\Gamma_1) \cap \mathcal{V}(\Gamma_2)\}$.

In the case that at least one theory is not convex, we guess a partition of α, $\phi = \{\alpha_1, \alpha_2, ..., \alpha_n\}$, which is expressed as $\{x = y \mid \text{for some } i : x, y \in \alpha_i\} \cup \{x \neq y \mid \text{for some } i, j : i \neq j, x \in \alpha_i, y \in \alpha_j\}$. If there is no partition ϕ such that both $\Gamma_1 \wedge \phi$ and $\Gamma_2 \wedge \phi$ are T-satisfiable, then the formula is not satisfiable in $T_1 + T_2$. Because the process of guessing is so time consuming, if we assume the time for a T_i solver to finish is $\mathcal{O}(T_i(n))$, then the overall complexity of this procedure is $\mathcal{O}(2^{n^2} \times (\mathcal{O}(T_1(n)) + \mathcal{O}(T_2(n))))$ [7].

In the other case that both theories are convex, complexity significantly decreases to $\mathcal{O}(n^4 \times (\mathcal{O}(T_1(n)) + \mathcal{O}(T_2(n))))$ [7] since instead of guessing, we can deduce the correct partition to use. The deduction process is taking terms of the form $x = y$ where $x, y \in \alpha$ and $T_1 \cup \Gamma_1 \models x = y$ (T-solver invoked here) and adding the equality to Γ_2. The same is done for equalities in Γ_2. Since the formula is necessarily finite, this process of propagating equalities terminates, and the individual T-solvers are invoked once more to check the satisfiability of the resulting sets. The convexity comes into play when sharing equalities since theory T_1 can assume $x^{\mathcal{M}_2} \neq y^{\mathcal{M}_2}$ when $T_2 \not\models x = y$ ($\mathcal{M}_2$ is the partial assignment for Γ_2).

A newer, experimentally illustrated (significantly) faster method of theory combination is based on incrementally constructing models for Γ_i and propagating equalities that are more intelligently guessed, using a DPLL search to backtrack on failure [7]. A large problem with the Nelson-Oppen method is that equalities are unnecessarily exhaustively propagated between theories. Since this method's process is based off smart guessing and not the structure of the theories themselves, it

[3]Private discussion with Sandip Ray

necessarily works for both convex and non-convex theories. The idea of the procedure is to
(i) Maintain models of each theory as the search progresses (on-line SMT algorithm)
(ii) Periodically check if both models agree in their interpretations that $u = v$, then case split on the equality (in the affirmative first)
(iii) Allow theory solvers to update their models in order to satisfy newly assigned literals or to imply fewer equalities.

In order to prevent infinite loops checking the same equality, we case split on $u = v$ only if $(u = v) \notin \mathcal{L}$, and then add it ($\mathcal{L}$ exists only for this purpose). Depending on the theory, some equalities can be deduced without invoking the solver (example: linear integer arithmetic [7]) and are exhaustively propagated to optimize performance. Other optimizations include postponing case splits on equalities until all others have completed and making sure that updated models never decrease the number of equivalence classes of variables in each model (called *diversifying* the model).

11. Drawbacks to SMT-based Verification

After seeing a great deal of expressive logics and efficient methods to decide them in this paper, we must step back and consider what the caveats are of dealing only with decidable theories. First of all, as we saw in the theory of interpreted sets and bounded quantification (call it ISBQ), loops must have their invariants discovered beforehand, which is most typically the most difficult process of verifying programs. In the code-test-iterate method of developing programs, annotating software with formal specifications is cumbersome and error-prone, and even supposing that the specification is correct, the current state of SMT solvers can take anywhere between a few seconds to many hours to decide if one's code models the specification severely interrupts the development process. The exponential nature of the solvers will almost certainly remain (at least for the foreseeable future) a constant hindrance to real-time correctness checking, or possibly even daily correctness checking.

12. Conclusion

As was presented in this paper, quite a few properties of a program can be verified using carefully constructed decidable theories. This is a fantastic tool that can be used for different static analyses and to add more automation to automated theorem proving. Indeed, the theory of bit vectors extends the limits of bounded model checking a formidable amount [17]. Further research into better heuristics for the solver itself will increase the effectiveness of SMT in its already established areas of dominance. Moreover, further research into more expressive decidable theories will allow more programs to be verified with more automation. As we say now, cycles are cheap but people are expensive - higher automation is key to making the use of formal methods a standard practice for commercial software developers, as they can spend less on finding bugs by simply letting a farm of computers do it for them.

References

[1] R. J. J. Bayardo and R. C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208, Providence, Rhode Island, 1997. AAAI, AAAI Press / The MIT Press.

[2] Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *VLSI Design*, pages 741–746, 2002.

[3] J. R. Levitt C. W. Barrett, D. L. Dill. A decision procedure for bit-vector arithmetic. In *Design Automation Conference, 1998. Proceedings*, pages 522–527, June 1998.

[4] M. Ganai A. Gupta C. Wang, F. Ivani. Deciding separation logic formulae by sat and incremental negative cycle elimination. In G. Sutcliff and A. Voronkov, editors, *LNCS 3835*, pages 322–336, Berlin, Germany, 2005. LPAR, Springer, Heidelberg.

[5] S. Cotton. Satisfiability checking with difference constraints. Master's thesis, Saarbruecken, May 2005.

[6] David Cyrluk, M. Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1997.

[7] L. de Moura and N. Bjørner. Model-based theory combination. volume 198, pages 37–49, Amsterdam, The Netherlands, The Netherlands, May 2008. SMT 2007, Elsevier Science Publishers B. V.

[8] E. I. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. In *DATE*, pages 142–149, 2002.

[9] R. Nieuwenhuis A. Oliveras C. Tinelli H. Ganzinger, G. Hagen. DPLL(T): Fast Decision Procedures. In R. Alur and D. Peled, editors, *16th International Conference on Computer Aided Verification, CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[10] B. Selman H. Kautz. Planning as satisfiability. In *ECAI 92*, pages 359–379. Tenth European Conference on Artificial Intelligence, August 1992.

[11] N. Shankar L. de Moura, B. Dutertre. A tutorial on satisfiability modulo theories. In W. Damm and H. Hermanns, editors, *LNCS 4590*, pages 20–36, Berlin, Germany, 2007. CAV, Springer, Heidelberg.

[12] D. Loveland M. Davis, G. Logemann. A machine program for theorem-proving, 1962.

[13] H. Putnam M. Davis. A computing procedure for quantification theory, 1960.

[14] R. Nieuwenhuis and A. Oliveras. Congruence closure with integer offsets. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850, pages 78–90, Berlin, Germany, 2003. LPAR, Springer, Heidelberg.

[15] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In J. Giesl, editor, *16th International Conference on Term Rewriting and Applications, RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer Heidelberg, 2005.

[16] F. Many R. Bjar. Solving the round robin problem using propositional logic. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence*, Austin, Texas, USA, 2000. AAAI/IAAI, AAAI Press / The MIT Press.

[17] A. Franzen A. Griggio Z. Hanna A. Nadel A. Palti R. Sebastiani R. Bruttomesso, A. Cimatti. A lazy and layered smt(bv) solver for hard industrial verification problems. In W. Damm and H. Hermanns, editors, *LNCS 4590*, pages 547–560, Berlin, Germany, 2007. CAV, Springer, Heidelberg.

[18] C. Tinelli R. Nieuwenhuis, A. Oliveras. Solving sat and sat modulo theories: from an abstract davis-putnam-logemann-loveland procedure to dpll(t), 2006.

[19] S. Qadeer S. K. Lahiri. Back to the future: Revisiting precise program verification using smt solvers. In Philip Wadler George C. Necula, editor, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL, 2008.

[20] J. P. M. Silva. *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*. PhD thesis, The University of Michigan, 2005.

[21] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *LNCS 2392*, pages 313–331, Berlin, Germany. Springer, Heidelberg.