DOI:10.1145/1995376.1995394

Checking the satisfiability of logical formulas, SMT solvers scale orders of magnitude beyond custom ad hoc solvers.

BY LEONARDO DE MOURA AND NIKOLAJ BJØRNER

Satisfiability Modulo Theories: Introduction and Applications

Constraint-satisfaction problems arise in diverse application areas, including software and hardware verification, type inference, static program analysis, test-case generation, scheduling, planning, and graph problems, and share a common trait—a core component using logical formulas for describing states and transformations between them. The most well-known constraint satisfaction problem is propositional satisfiability, or SAT, aiming to decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing true/false values for its variables. Some problems are more naturally described with richer languages (such as arithmetic). A supporting theory (of arithmetic) is then required to capture the meaning of the formulas. Solvers for such formulations are commonly called "satisfiability modulo theories," or SMT, solvers.

In the past decade, SMT solvers have attracted increased attention due to technological advances and industrial applications. Yet SMT solvers draw on some of the most fundamental areas of computer science, as well as a century of symbolic logic. They combine the problem of Boolean satisfiability

» key insights

- **Many tools for program analysis, testing, and verification are based on mathematical logic as the calculus of computation.**
- **SMT solvers are the core engine of many of these tools.**
- **Modern SMT solvers integrate specialized solvers with propositional satisfiability search techniques.**

Figure 1. Encoding job-shop scheduling.

$d_{i,j}$	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

max = 8

Solution

$t_{1,1} = 5,\quad t_{1,2} = 7,$
$t_{2,1} = 2,\quad t_{2,2} = 6,$
$t_{3,1} = 0,\quad t_{3,2} = 3$

Encoding

$$\begin{array}{l}
(t_{1,1} \geq 0) \wedge (t_{1,2} \geq t_{1,1} + 2) \wedge (t_{1,2} + 1 \leq 8) \wedge \\
(t_{2,1} \geq 0) \wedge (t_{2,2} \geq t_{2,1} + 3) \wedge (t_{2,2} + 1 \leq 8) \wedge \\
(t_{3,1} \geq 0) \wedge (t_{3,2} \geq t_{3,1} + 2) \wedge (t_{3,2} + 3 \leq 8) \wedge \\
((t_{1,1} \geq t_{2,1} + 3) \vee (t_{2,1} \geq t_{1,1} + 2)) \wedge \\
((t_{1,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{1,1} + 2)) \wedge \\
((t_{2,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{2,1} + 3)) \wedge \\
((t_{1,2} \geq t_{2,2} + 1) \vee (t_{2,2} \geq t_{1,2} + 1)) \wedge \\
((t_{1,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{1,2} + 1)) \wedge \\
((t_{2,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{2,2} + 1))
\end{array}$$

with domains (such as those studied in convex optimization and term-manipulating symbolic systems). They involve the decision problem, completeness and incompleteness of logical theories, and complexity theory. Here, we explore the field of SMT and some of its applications.

Increased attention has led to enormous progress in constraint-satisfaction problems that can be solved due to innovations in core algorithms, data structures, heuristics, and the careful use of modern microprocessors. Modern SAT[27] procedures can check formulas with hundreds of thousands of variables. Similar progress has been observed for SMT solvers for more commonly occurring theories, including such state-of-the art SMT solvers as Barcelogic,[8] CVC,[3,7] MathSAT,[10] Yices,[18] and Z3.[14]

The annual competitions for SAT (http://www.satcompetition.org) and SMT (http://www.smtcomp.org) are a key driving force.[4] An important ingredient is a common interchange format for benchmarks, called SMT-LIB,[33] and the classification of benchmarks into various categories, depending which theories are required. Conversely, a growing number of applications can generate benchmarks in the SMT-LIB format to further improve SMT solvers.

There is a relatively long tradition dating to the late-1970s of using SMT solvers in specialized contexts. One prolific case is theorem-proving systems (such as ACL2[26] and PVS[32]) that use decision procedures to discharge lemmas encountered during interactive proofs. SMT solvers have also been used for the past 15 years in the context of program verification and extended static checking[21] where verification focuses on assertion checking.

Progress in the past four years in SMT solvers has enabled their use in diverse applications, including interactive theorem provers and extended static checkers, as well as in scheduling, planning, test-case generation, model-based testing and program development, static program analysis, program synthesis, and run-time analysis.

We begin by introducing an application we use as a running example.

Scheduling. Consider the classical job-shop-scheduling decision problem, involving n jobs, each composed of m tasks of varying duration that must be performed consecutively on m machines. The start of a new task can be delayed as long as needed in order for a machine to become available, but tasks cannot be interrupted once they are started. The problem involves essentially two types of constraints:

Precedence. Between two tasks in the same job; and

Resource. Specifying that no two different tasks requiring the same machine are able to execute at the same time.

Given a total maximum time *max* and the duration of each task, the problem consists of deciding whether there is a schedule such that the end-time of every task is less than or equal to *max* time units. We use $d_{i,j}$ to denote the duration of the j-th task of job i. A schedule is specified by the start-time ($t_{i,j}$) for the j-th task of every job i. The job-shop-scheduling problem can be encoded in SMT using the theory of linear arithmetic. A precedence constraint between two consecutive tasks $t_{i,j}$ and $t_{i,j+1}$ is encoded using the inequality $t_{i,j+1} \geq t_{i,j} + d_{i,j}$; this inequality states that the start-time of task $j + 1$ must be greater than or equal to the start time of task j plus its duration. A resource constraint between two tasks from different jobs i and i' requiring the same machine j is encoded using the formula $(t_{i,j} \geq t_{i',j} + d_{i',j}) \vee (t_{i',j} \geq t_{i,j} + d_{i,j})$, stating the two tasks do not overlap. The start time of the first task of every job i must be greater than or equal to zero, so the result is $t_{i,1} \geq 0$. Finally, the end time of the last task must be less than or equal to *max*, hence $t_{i,m} + d_{i,m} \leq max$. Figure 1 is an instance of the job-scheduling problem, its encoding as a logical formula, and a solution. The logical formula combines logical connectives (conjunctions, disjunction, and negation) with atomic formulas in the form of linear arithmetic inequalities. We call it an SMT formula. The solution in Figure 1 is a satisfying assignment, a mapping from variables $t_{i,j}$ to values that make the formula *true*.

SMT-Solving Techniques

Modern SMT solvers use procedures for deciding the satisfiability of conjunctions of literals, where a literal is an atomic formula or the negation of an atomic formula. Throughout this article, we call these procedures "theory solvers." The scheduling application demonstrates that this kind of procedure alone is not sufficient in practice, because the encoding contains disjunctive sub-formulas, as in

$$(t_{1,1} \geq t_{2,1} + 3) \vee (t_{2,1} \geq t_{1,1} + 2)$$

SMT solvers handle sub-formulas like this by performing case analysis, which is in the core of most automated deduction tools. Most SMT solvers rely on efficient satisfiability procedures for propositional logic (SAT solvers) for performing case analysis efficiently. A standard technique for integrating SAT solvers and theory solvers[1,5,15,20,30] is described next.

SAT: A propositional core. Propositional logic is a special case of predicate logic in which formulas are built from Boolean variables, called atoms, and composed using logical connectives (such as conjunction, disjunction, and negation). The satisfiability problem for propositional logic is famously known as an NP-complete problem[12] and therefore in principle computationally intractable. Yet recent advances in efficient propositional logic algorithms have moved the boundaries for what is intractable when it comes to practical applications.[27]

Most successful SAT solvers are based on an approach called "systematic search." The search space is a tree with each vertex representing a Boolean variable and the out edges representing the two choices (*true* and *false*) for this variable. For a formula containing n Boolean variables, the tree has 2^n leaves. Each path from the root to a leaf corresponds to a truth assignment. A model is a truth assignment that makes the formula true. We also say the model satisfies the formula, and the formula is satisfiable.

Most search-based SAT solvers are based on the DPLL/Davis-Putnam-Logemann-Loveland algorithm.[13] The DPLL algorithm tries to build a model using three main operations: `decide`, `propagate`, and `backtrack`. The algorithm benefits from a restricted representation of formulas in conjunctive normal form, or CNF. CNF formulas are restricted to be conjunctions of clauses, with each clause, in turn, a disjunction of literals. Recall that a literal is an atom or the negation of an atom; for example, the formula $\neg p \wedge (p \vee q)$ is in CNF. The operation `decide` heuristically chooses an unassigned atom, assigning it to *true* or *false*, and is also called branching or case-splitting. The operation `propagate` deduces the consequences of a partial truth assignment using deduction rules. The most widely used deduction rule is the unit-clause rule, stating that if a clause has all but one literal assigned to *false* and the remaining literal l is unassigned, then the only way for the clause to evaluate to true is to assign l to *true*.

Let C be the clause $p \vee \neg q \vee \neg r$, and M the partial truth assignment $\{p \rightarrow false, r \rightarrow true\}$, then the only way for C to evaluate to *true* is by assigning q to *false*. Given a partial truth assignment M and a clause C in the CNF formula, such that all literals of C are assigned to *false* in M, then there is no way to extend M to a complete model M' that satisfies the given formula. We say this is a conflict, and C is a conflicting clause. A conflict indicates some of the earlier decisions cannot lead to a truth assignment that satisfies the given formula, and the DPLL procedure must `backtrack` and try a different branch value. If a conflict is detected and there are no decisions to backtrack, then the formula is unsatisfiable; that is, it does not have a model. Many significant improvements to this basic procedure have been proposed over the years, with the main ones being lemma learning, non-chronological backtracking, and efficient indexing techniques for applying the unit-clause rule and preprocessing techniques.[27]

A solver for difference arithmetic. The job-shop-scheduling decision problem can be solved by combining a SAT solver with a theory solver for difference arithmetic. Difference arithmetic is a fragment of linear arithmetic, where predicates are restricted to be of the form $t - s \leq c$ and where t and s are variables and c a numeric constant (such as 1 and 3). Every atom in Figure 1 can be put into this form; for example, the atom $t_{3,1} \geq t_{2,1}+3$ is equivalent to the atom $t_{2,1}-t_{3,1} \leq -3$. For atoms of the form $s \leq c$ and $s \geq c$, a special fresh variable z is used. We say z is the zero variable, and the atoms are represented in difference arithmetic as $s - z \leq c$ and $z - s \leq -c$, respectively; for example, the atom $t_{3,2} + 3 \leq 8$ is represented in difference arithmetic as $t_{3,2} - z \leq 5$. A set of difference arithmetic atoms can be checked efficiently for satisfiability by searching for negative cycles in weighted directed graphs. In the graph representation, each variable corresponds to a node, and an inequality of the form $t - s \leq c$ corresponds to an edge from s to t with weight c. Figure 2 is a subset of atoms (in difference arithmetic form) from the example in Figure 1, along with the corresponding graph. The negative cycle, with weight –2, is shown by dashed lines. The cycle corresponds to the following schedule that cannot be completed in eight time units:

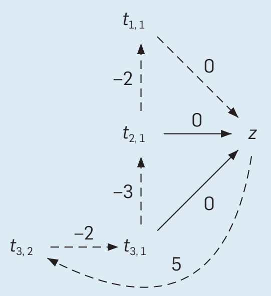
task 1/job 1 → task 1/job 2 →
task 1/job 3 → task 2/job 3

Recall that the scheduling problem in Figure 1 is satisfiable but requires assigning a different combination of atoms to *true*.

Interfacing solvers with SAT. We've outlined a theory solver for difference arithmetic and now describe how a SAT procedure interacts with this theory solver. The key idea is to create an abstraction that maps the atoms in an SMT formula into fresh Boolean variables $p_1, \ldots, p_n$; for example, the formula $\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5)$ is translated into $\neg p_1 \wedge (p_1 \vee p_2)$, where the atoms $a \geq 3$ and $a \geq 5$ are replaced by the Boolean variables p_1 and p_2, respectively. The new abstract formula can then be processed by a regular SAT procedure. If the SAT procedure finds the abstract formula to be unsatisfiable, then so, too, is the SMT formula. On the other hand, if the abstract formula is found to be satisfiable, the theory solver is used to check the model produced by the SAT procedure. The idea is that any model produced by the SAT procedure induces a set of literals; for example, $\{p_1 \rightarrow \text{false}, p_2 \rightarrow \text{true}\}$ is a model for the formula $\neg p_1 \wedge (p_1 \vee p_2)$, inducing the set of literals $\{\neg(a \geq 3), a \geq 5\}$ that is unsatisfiable in the theory of arithmetic. Therefore, the formula (clause) $a \geq 3 \vee \neg(a \geq 5)$ is valid in the theory of arithmetic. The abstraction of this formula is the clause $p_1 \vee \neg p_2$. We say it is a "theory lemma," and since it is based on a valid formula from the theory of arithmetic, we can then add it to our original formula, obtaining the new formula:

Figure 2. Example of difference arithmetic.

$$\begin{array}{lcccr}
z & - & t_{1,1} & \leq & 0 \\
z & - & t_{2,1} & \leq & 0 \\
z & - & t_{3,1} & \leq & 0 \\
t_{3,2} & - & z & \leq & 5 \\
t_{3,1} & - & t_{3,2} & \leq & -2 \\
t_{2,1} & - & t_{3,1} & \leq & -3 \\
t_{1,1} & - & t_{2,1} & \leq & -2
\end{array}$$



$$\neg p_1 \wedge (p_1 \vee p_2) \wedge (p_1 \vee \neg p_2)$$

The SAT solver is executed again, taking the new formula as input, and finds the new formula to be unsatisfiable, proving the original formula $\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5)$ is also unsatisfiable. In practice, many theory lemmas are created until this process converges. Note, too, this process always converges because there is a finite number of atoms, and, consequently, there is a finite number of theory lemmas that can be created using them.

Given an unsatisfiable set of theory literals S, we say a justification for S is any unsatisfiable subset J of S. Any unsatisfiable set S is, of course, also a justification for itself. We say a justification J is non-redundant if there is no strict subset J' of J that is also unsatisfiable. It is desirable to have a theory solver that produces non-redundant justifications, as they may drastically reduce the search space. This observation follows from the fact that smaller sets produce smaller theory lemmas (clauses) and consequently have fewer satisfying assignments.

Returning to the example in Figure 2, the negative cycle corresponds to a non-redundant unsatisfiable set of difference atoms. The negation of these atoms corresponds to the following valid clause in difference arithmetic:

$$\neg(t_{3,1} - t_{3,2} \leq -2) \vee \neg(t_{2,1} - t_{3,1} \leq -3) \vee$$
$$\neg(t_{1,1} - t_{2,1} \leq -2) \vee \neg(z - t_{1,1} \leq 0) \vee$$
$$\neg(t_{3,2} - z \leq 5)$$

This integration scheme is also known as the "lazy offline" approach and includes many refinements; one is to have a tighter integration between the two procedures, where the theory solver is used to check partial truth assignments being explored by the SAT solver (online integration). In it, additional performance gains can be obtained if the theory solver is incremental (new constraints can be added at minimal cost) and backtrackable (constraints can be removed at minimal cost). Theory deduction rules can also be used to prune the search space being explored by the DPLL solver (theory propagation). In difference arithmetic, theory propagation can be implemented by computing the shortest distance between two nodes. Returning to the example in Figure 2, assume the inequality $t_{2,1} - t_{3,1} \leq -3$ is not there. Thus, the graph on the right-hand side will not contain an edge from $t_{3,1}$ to $t_{2,1}$ and, consequently, the negative cycle. The shortest distance between the nodes $t_{2,1}$ and $t_{3,1}$ is 1 by following the path

$$t_{2,1} \rightarrow t_{1,1} \rightarrow z \rightarrow t_{3,2} \rightarrow t_{3,1}$$

This fact implies that $t_{3,1} - t_{2,1} \leq 1$, and one can verify the result by adding the inequalities associated with each edge. The inequality $t_{3,1} - t_{2,1} \leq 1$ is equivalent to $t_{2,1} - t_{3,1} \geq -1$, implying $\neg(t_{2,1} - t_{3,1} \leq -3)$. Therefore, if the SAT solver has assigned the atoms $t_{1,1} - t_{2,1} \leq -2$, $z - t_{1,1} \leq 0$, $t_{3,2} - z \leq 5$ and $t_{3,1} - t_{3,2} \leq -2$ to *true*, then, by theory propagation, the atom $t_{2,1} - t_{3,1} \leq -3$ can be assigned to *false*, thus avoiding the inconsistency (negative cycle) in Figure 2.

Program 3.1. Greatest common divisor program.

```
int GCD (int x, int y)
   while (true) {
      int m = x % y;
      if (m == 0) return y;
      x = y;
      y = m;
   }
}
```

Program 3.2. Greatest common divisor path formula.

```
int GCD (int x0, int y0) {
   int m0 = x0 % y0;         (m0 = x0 % y0)    ∧
   assert (m0 != 0);         ¬(m0 = 0)         ∧
   int x1 = y0;              (x1 = y0)         ∧
   int y1 = m0;              (y1 = m0)         ∧
   int m1 = x1 % y1;         (m1 = x1 % y1)    ∧
   assert (m1 == 0);         (m1 = 0)
}
```

SMT in Software Engineering

Software developers use logical formulas to describe program states and transformations between program states, a procedure at the core of most software-engineering tools that analyze, verify, or test programs. Here, we describe a few such applications:

Dynamic symbolic execution. SMT solvers play a central role in dynamic symbolic execution. A number of tools used in industry are based on dynamic symbolic execution, including CUTE, Klee, DART, SAGE, Pex, and Yogi,[23] designed to collect explored program paths as formulas, using solvers to identify new test inputs that can steer execution into new branches. SMT solvers are a good fit for symbolic execution because the semantics of most program statements are easily modeled using theories supported by these solvers. We later introduce the various theories that are used, but here we focus on connecting constraints with a solver. To illustrate the basic idea of dynamic symbolic execution, consider the greatest common divisor in Program 3.1, taking the inputs x and y and producing the greatest common divisor of x and y.

Program 3.2 represents the static single assignment unfolding corresponding to the case where the loop is exited in the second iteration. Assertions are used to enforce that the condition of the if statement is not satisfied in the first iteration and is in the second iteration. The sequence of instructions is equivalently represented as a formula where the assignment statements have been turned into equations.

The resulting path formula is satisfiable. One satisfying assignment that can be found using an SMT solver is of the form:

$$x_0 = 2, y_0 = 4, m_0 = 2, x_1 = 4, y_1 = 2, m_1 = 0$$

It can be used as input to the original program; in this example, the call `GCD(2,4)` causes the loop to be entered twice, as expected.

Fuzz testing is a software-testing technique that provides invalid or unexpected data to a program. The program being fuzzed is opaque, and fuzzing is performed by perturbing input vectors using random walks. "White-box fuzzing" combines fuzz testing and dynamic symbolic execution and is actively used at Microsoft. Complementing traditional fuzz testing, it has been instrumental in uncovering several subtle security-critical bugs that traditional testing methods are unable to find.

Program model checking. Dynamic symbolic execution finds input that can guide execution into bugs. This method alone does not guarantee that programs are free of all the errors being checked for. The goal of program model checking tools is to automatically check for freedom from selected categories of errors. The idea is to explore all possible executions using a finite and sufficiently small abstraction of the program state space. The tools BLAST,[25] SDV,[2] and SMV from Cadence[a] perform program model checking. Both SDV and SMV are used as part of commercial tool offerings. The program fragment in Program 3.3 is an example of finite-state abstraction, accessing requests using `GetNextRequest`. The call is protected by a lock. A question is whether it is possible to exit the loop without having a lock. The program has a very large, potentially unbounded, number of states, since the value of the program variable count can grow arbitrarily.

However, from the point of view of locking, the actual values of count and `old _ count` are not interesting. On the other hand, the relationship between these program variables contains useful information. Program 3.4 is a finite-state abstraction of the same locking program. The Boolean variable `b` encodes the relation `count == old _ count`. In it, we use the symbol $*$ to represent a Boolean expression that nondeterministically evaluates to *true* or *false*. The abstract program contains only Boolean variables, thus a finite number of states. We can now explore the finite number of branches of the abstract program to verify the lock is always held when exiting the loop.

SMT solvers are used for constructing finite-state abstractions, like the one in Program 3.4. Abstractions can be created through several approaches; in one, each statement in the program

a http://www.kenmcmil.com

Most SMT solvers rely on efficient satisfiability procedures for propositional logic (SAT solvers) for performing case analysis efficiently.

is individually abstracted; for example, consider the statement `count = count + 1`. The abstraction of it is essentially a relation between the current and the new values of the Boolean variable `b`. SMT solvers are used to compute the relation by proving theorems, as in

```
count == old _ count →
count+1 != old _ count
```

which is equivalent to checking unsatisfiability of the negation

```
count == old _ count ∧
count+1 == old _ count
```

The theorem says if the current value of `b` is *true*, then after executing the statement `count = count + 1`, the value of `b` will be *false*. Note that if `b` is *false*, then neither of the following conjectures is valid:

```
count != old _ count →
count+1 == old _ count
count != old _ count →
count+1  != old _ count
```

In each, an SMT solver will produce a model for the negation of the conjec-

Program 3.3. Processing requests using locks.

```
do {
    lock ();
    old_count = count;
    request = GetNextRequest();
    if (request != NULL) {
        unlock();
        ProcessRequest(request);
        count = count + 1;
    }
}
while (old_count != count);
unlock();
```

Program 3.4. Processing requests using locks, abstracted.

```
do {
    lock ();
    b = true;
    request = GetNextRequest();
    if (request != NULL) {
        unlock();
        ProcessRequest(request);
        if (b) b = false; else b = *;
    }
}
while (!b);
unlock();
```

ture. Therefore, the model is a counterexample of the conjecture, and when the current value of `b` is false, nothing can be said about its value after the execution of the statement. The result of these three proof attempts is then used to replace the statement `count = count + 1;` by `if (b) b = false; else b = *;`. A finite state model checker can now be used on the Boolean program and will establish that `b` is always *true* when control reaches this statement, verifying that calls to `lock()` are balanced with calls to `unlock()` in the original program.

Static program analysis. Static program analysis tools work like dynamic-symbolic-execution tools, checking feasibility of program paths. On the other hand, they never require executing programs and can analyze software libraries and utilities independently of how they are used. One advantage of using modern SMT solvers in static program analysis is they accurately capture the semantics of most basic operations used by mainstream programming languages. The program fragment in Program 3.5 illustrates the need for static program analysis to use bit-precise reasoning, searching for an index in a sorted array `arr` containing a key.

The `assert` statement is a precondition for the procedure, restricting the input to fall within the bounds of the array `arr`. The program performs several operations involving arithmetic, so a theory and corresponding solver that understands arithmetic is arguably a good match. However, it is important for software-analysis tools to take into account that languages (such as Java, C#, and C/C++) all use fixed-width bit-vectors as representation for values of type `int`, meaning the accurate theory for `int` is two-complements modular arithmetic. Assuming a bit-width of 32b, the maximal positive 32b integer is $2^{31}–1$, and the smallest negative 32b integer is $–2^{31}$. If both `low` and `high` are 2^{30}, `low + high` evaluates to 2^{31}, which is treated as the negative number $–2^{31}$. The presumed assertion $0 \le mid < high$ does therefore not hold. Fortunately, several modern SMT solvers support the theory of "bit-vectors," accurately capturing the semantics of modular arithmetic. The bug does not escape an analysis based on the theory of bit-vectors. Such analysis would check that the array read `arr[mid]` is within bounds during the first iteration by checking the formula

$$(low > high \vee 0 \le low < high < arr.length) \wedge (low \le high \rightarrow 0 \le (low + high)/2 < arr.length)$$

As in the case of code fragment 3.5, the formula is not valid. The values `low = high =` 2^{30}, `arr.length =` $2^{30}+1$ provide a counterexample. The use of SMT solvers for bit-precise static-analysis tools is an active area of research and development in Microsoft Research. Integration with the solver Z3[14] and the static analysis tool PREfix led to the automatic discovery of several overflow-related bugs in Microsoft's codebase.

Program verification. The ideal of verified software is a long-running quest since Robert Floyd and C.A.R. Hoare introduced (in the late 1960s) program verification by assigning logical assertions to programs. Extended static checking uses the methods developed for program verification but in the more limited context of checking absence of runtime errors. The SMT solver Simplify[16] was developed in the context of the extended static-checking systems ESC/Modula 3 and ESC/Java.[21] This work was and continues to be the inspiration for several subsequent verification tools, including Why[19] and Boogie.[3] These systems are actively used as bridges from several different front ends to SMT-solver back ends; for example, Boogie is used as a back end for systems that verify code from languages (such as an extended version of C# called Spec#), as well as low-level systems code written in C. Current practice indicates that a lone software developer can drive these tools to verify properties of large codebases with several hundred thousand lines of code. A more ambitious project is the Verifying C-Compiler system,[11] targeting functional correctness properties of Microsoft's Viridian Hyper-Visor. The Hyper-Visor is a relatively small (100,000 lines) operating-system layer, yet formulating and establishing correctness properties is a challenge. The entire verification effort for this layer is estimated by Microsoft to take around 60 programmer years.

Program-verification applications often use theories not already supported by existing specialized solvers but that are supported indirectly using axiomatizations with quantifiers. As an example of such a theory, in object-oriented-type systems used for Java and C#, it is the case that objects are related using a single inheritance scheme; that is, every object inherits from at most one unique immediate parent. To illustrate the theory, let *array-of*(x) be the array type constructor for arrays of values of type x. In some programming languages, if x is a subtype of y, then *array-of*(x) is a subtype of *array-*

Figure 3. Axioms for *sub*.

$(\forall x{:}\ sub(x, x))$

$(\forall x,y,z{:}\ sub(x, y) \wedge sub(y, z) \rightarrow sub(x, z))$

$(\forall x,y{:}\ sub(x, y) \wedge sub(y, x) \rightarrow x = y)$

$(\forall x,y,z{:}\ sub(x, y) \wedge sub(x, z) \rightarrow sub(y, z) \vee sub(z, y))$

$(\forall x,y{:}\ sub(x, y) \rightarrow sub(array\text{-}of(x), array\text{-}of(y)))$

Program 3.5. Binary search.

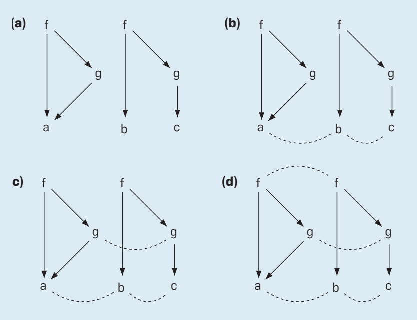
```
int binary_search(
    int[] arr, int low, int high, int key) {
        assert (low > high || 0 <= low < high);
        while (low <= high) {
             //Find middle value
             int mid = (low + high)/2;
             assert (0 <= mid < high);
             int val = arr[mid];
             //Refine range
             if (key == val) return mid;
             if (val > key) low = mid+1;
             else high = mid–1;
        }
        return –1;
}
```

of(y). In this case, we say arrays behave in a monotone way with respect to inheritance. Using first-order axioms, we specify in Figure 3 that the inheritance relation *sub*(x, y) is a partial order satisfying the single inheritance property and that the array type constructor *array-of*(x) is monotone with respect to inheritance.

The theory of object inheritance illustrates why SMT solvers targeted at expressive program analysis benefit from general support for quantifiers.

All the applications we have treated so far also rely on a fundamental theory we have not described: the theory of equality and free functions. The axioms used for object inheritance used the binary predicate *sub* and the function *array-of*. All we know about *array-of* is that it is monotone over sub, and, for this reason, we say the function is free. Decision procedures for free functions are particularly important because it is often possible to reduce decision problems to queries over free functions. Given a conjunction of equalities between terms using free functions, a congruence closure algorithm can be used to represent the smallest set of implied equalities. This representation can help check if a mixture of equalities and disequalities are satisfiable, checking that the terms on both sides of each disequality are in different equivalence classes. Efficient algorithms for computing congruence closure are the subject of long-running research[17] in which terms are represented as directed acyclic graphs, or DAGS. Figure 4 outlines the operation of a congruence closure algorithm on the following limited example
$a = b, b = c, f(a, g(a)) \neq f(b, g(c))$

Figure 4. Example of congruence closure.



SMT solvers are a good fit for symbolic execution because the semantics of most program statements are easily modeled using theories supported by these solvers.

In Figure 4(a), we spelled out a DAG for all terms in the example; in Figure 4(b), the equivalences $a = b$ and $b = c$ are represented by dashed lines; in Figure 4(c), nodes $g(a)$ and $g(c)$ are congruent because $a = c$ is implied by the first two equalities; and finally, in Figure 4(d), nodes $f(a, g(a))$ and $f(b, g(c))$ are also congruent, hence the example is unsatisfiable due to the required disequality $f(a, g(a)) \neq f(b, g(c))$.

Modeling. SMT solvers represent an interesting opportunity for high-level software-modeling tools. In some contexts these tools use domains from mathematics (such as algebraic datatypes, arrays, sets, and maps) and have also been the subject of long-running research in the context of SMT solvers. Here, we introduce the array domain that is frequently used in software modeling.

The theory of arrays was introduced by John McCarthy in a 1962 paper[28] as part of forming a broader agenda for a calculus of computation. It included two functions: *read* and *write*. The term *read*(a, i) produces the value of array a at index i, and the term *write*(a, i, v) produces an array equal to a, except for possibly index i, which maps to v. To make the terminology closer to how arrays are read in programs, we write $a[i]$ instead of *read*(a, i). These properties are summarized through two equations:

$$write(a, i, v)[i] = v$$
$$write(a, i, v)[j] = a[j] \text{ for } i \neq j$$

They state that the result of reading $write(a, i, v)$ at index j is v for $i = j$. Reading the array at any other index produces the same value as $a[j]$. Consider, for example, the program swap, swapping the entries $a[i]$ and $a[j]$.

```
void swap (int [] a, int i, int j)

{
int tmp = a[i];
a[i] = a[j];
a[j] = tmp;
}
```

The statement that $a[i]$ contains the previous value of $a[j]$can be expressed as
$a[j] = write(write(a, i, a[j]), j, a[i])[i]$

Here, we summarize a few areas in the context of software modeling where SMT solvers are used. Model programs are behavioral specifications that can be described succinctly and at a high level of abstraction. These descriptions are state machines that use abstract domains. SMT solvers are used to perform bounded model-checking of such descriptions. The main idea of bounded model-checking is to explore a bounded symbolic execution of a program or model. Thus, given a bound (such as 17), the transitions of the state machines are unrolled into a logical formula describing all possible executions using 17 steps. Model-based designs use high-level languages for describing software systems. Implementations are derived by refinements. Modeling languages present an advantage, as they allow software developers to explore a design space without committing all design decisions up front. SMT solvers are the symbolic reasoning engines used in model-based designs; for example, they are used for type-checking designs and in the search for different consistent choices. Model-based testing uses high-level models of software systems, including network protocols, to derive test oracles. SMT solvers have been used in this context for exploring related models using symbolic execution. Model-based testing is used on a large scale by Microsoft developers in the context of disclosure and documentation of Microsoft network protocols.[24] The model-based tools use SMT solvers for generating combinations of test inputs, as well as for performing symbolic exploration of models.

One advantage of using modern SMT solvers in static program analysis is they accurately capture the semantics of most basic operations used by mainstream programming languages.

Combining Theory Solvers

How to combine multiple theory solvers is a fundamental problem for SMT solvers. As we discussed earlier, applications ranging from test-case generation to software verification require a combination of theories; for example, a combination of arithmetic and arrays is needed to reason about Program 3.5. Fundamental questions include: Is the union of two decidable theories still decidable? Is the union consistent? And how can we combine different theory solvers? In general, combining theory solvers is a very difficult problem. However, useful special cases have good answers. An established framework for combining theory solvers is known as the Nelson-Oppen combination method,[29] which assumes theories do not share symbols except for the equality relation. When the only shared symbol is the equality relation, we say the theories are disjoint; for example, the theory of linear arithmetic uses the constants, functions, and relations $+$, 0, 1, $\leq$, and the theory of arrays uses the disjoint set *read*, *write*. It should also be possible to merge the models from the two theory solvers into one without contradicting assumptions one theory might have about the size of models. A condition that guarantees solutions can be combined is known as "stable infiniteness"; a theory T is stably infinite if whenever a (quantifier-free) formula is satisfiable in T, then it is satisfiable in a model of T with an infinite universe (size).

In many practical cases, the disjointness and stable infiniteness conditions are easily satisfied when combining theory solvers. However, not all theory combinations satisfy these side conditions, and research over the past 10 years has sought to generalize the framework where signatures are non-disjoint or where theories are non-stably infinite.[22,34]

Convexity, complexity, and propositional search. Convexity is an important notion in the context of combining theories. A theory is convex if for all sets of ground literals S and all sets of equalities between variables E if S implies the disjunction of E, then it also implies at least one equation of E; for example, the theory of free functions is convex, but difference arithmetic over integers is not.

Convexity plays an important role in operations research, as well as in SMT, because efficient, polynomial time techniques exist for combining solvers for convex theories.[31] The key property is that the equalities can be deduced, without backtracking, instead of guessed, with backtracking. On the other hand, nonconvex theories incur a potential exponential time combination overhead. It therefore becomes an additional requirement on solvers in the Nelson-Oppen combination method that they also indicate which variables are implied equal based on a set of assertions.

The advent in the late-1990s of efficient methods for propositional search allowed viewing the theory combination problem from a different, more advantageous perspective. The delayed theory combination[9] method creates one atomic equality for every pair of variables shared between solvers. These additional atomic equalities are assigned to *true* or *false* by a SAT solver. In this approach, the SAT solver is used to guess the correct equalities between shared variables. If the theory solvers disagree with the (dis)equalities, then the conflict causes the SAT solver to backtrack. The approach is oblivious to whether or not theories are convex. Delayed theory combination potentially pollutes the search space with a large number of mostly useless new atomic equalities. The "Model-based theory combination" method[14] allows more efficient handling of convex and non-convex theories, asking the solvers to generate a model. The atomic equality predicates are created only if two shared variables are equal in a model.

Conclusion

Over the past 10 years, SMT has become the core engine behind a range of powerful technologies and an active, exciting area of research with many practical applications. We have presented some of the basic ideas but did not cover many details and heuristics; other recent topics in SMT research[6] include proof-checking, integration with first-order quantifiers, quantifier elimination methods, and extraction of so-called Craig interpolant formulas from proofs. We also did not cover several existing and emerging applications, including sophisticated run-time analysis of real-time embedded systems,[b] estimating asymptotic run-time bounds of programs, and program synthesis.

SMT-solving technologies have had a positive effect on a number of application areas, providing rich feedback in terms of experimental data. The progress in the past six years has relied heavily on experimental evaluations that uncovered new theoretical challenges, including better representations and algorithms, efficient methods for combining procedures, theories for quantifier reasoning, and various extensions to the basic search method. C

b http://www.eecs.berkeley.edu/~sseshia/research/embedded.html

References

1. Audemard, G., Bertoli, P., Cimatti, A., Kornilowicz, A., and Sebastiani, R. A SAT-based approach for solving formulas over Boolean and linear mathematical propositions. In *Proceedings of the Conference on Automated Deduction, Vol. 2392 of LNCS* (Copenhagen, July 27–30). Springer-Verlag, Berlin, 2002.
2. Ball, T. and Rajamani, S.K. The SLAM project: Debugging system software via static analysis. (Symposium on Principles of Programming Languages). *SIGPLAN Notices 37*, 1 (Jan. 16–18, 2002), 1–3.
3. Barnett, M., Leino, K.R.M., and Schulte, W. The Spec# programming system: An overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices, LNCS 3362* (Marseille, Mar. 10–13). Springer-Verlag, Berlin, 2005, 49–69.
4. Barrett, C., de Moura, L., and Stump, A. Design and results of the first Satisfiability Modulo Theories Competition. *Journal of Automated Reasoning 35*, 4 (Nov. 2005), 372–390.
5. Barrett, C., Dill, D., and Stump, A. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the International Conference on Computer Aided Verification* (Copenhagen, July, 27–31). Springer-Verlag, Berlin 2002, 236–249.
6. Barrett, C., Sebastiani, R., Seshia, S.A., and Tinelli, C. *Satisfiability Modulo Theories, Vol. 185 of Frontiers in Artificial Intelligence and Applications*, Chapter 26. IOS Press, Feb. 2009, 825–885.
7. Barrett, C. and Tinelli, C. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification, Vol. 4590 of LNCS*, W. Damm and H. Hermanns, Eds. (Berlin, July 3–7). Springer-Verlag, Berlin, 2007, 298–302.
8. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez Carbonell, E., and Rubio, A. The Barcelogic SMT Solver. In *Proceedings of the 20th International Conference on Computer Aided Verification, Vol. 5123 of LNCS*, A. Gupta and S. Malik, Eds. (Princeton, July 7–14). Springer-Verlag, Berlin, 2008, 294–298.
9. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., Ranise, S., van Rossum, P., and Sebastiani, R. Efficient satisfiability modulo theories via delayed theory combination. In *Proceedings of the International Conference on Computer Aided Verification, Vol. 3576 of LNCS*, K. Etessami and S. K. Rajamani, Eds. (Edinburgh, July 6–12). Springer-Verlag, Berlin, 2005, 335–349.
10. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., and Sebastiani, R. The MathSAT 4 SMT Solver. In *Proceedings of the 18th International Conference on Computer Aided Verification, Vol. 5123 of LNCS*. Springer-Verlag, Berlin, 2008.
11. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics* (Munich, Aug. 17–20). Springer-Verlag. Berlin, 2009, 23–42.
12. Cook, S.A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (May 3–5). ACM Press, New York, 1971, 151–158.
13. Davis, M., Logemann, G., and Loveland, D. A machine program for theorem proving. *Commun. ACM 5*, 2 (July 1962), 394–397.
14. de Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the International Conference on tools and algorithms for the Construction and Analysis of Systems, Vol. 4963 of LNCS*, C.R. Ramakrishnan and J. Rehof, Eds. (Budapest, Mar. 29–Apr. 6). Springer-Verlag, Berlin, 2008, 337–340.
15. de Moura, L. and Rueß, H. Lemmas on demand for satisfiability solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing* (Cincinnati, May 6–9, 2002).
16. Detlefs, D., Nelson, G., and Saxe, J.B. Simplify: A theorem prover for program checking. *Journal of the ACM 52*, 3 (May 2005), 365–473.
17. Downey, P.J., Sethi, R., and Tarjan, R.E. Variations on the common subexpression problem. *Journal of the ACM 27*, 4 (Oct. 1980), 758–771.
18. Dutertre, B. and de Moura, L. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 16th International Conference on Computer Aided Verification, Vol. 4144 of LNCS* (Seattle, Aug. 17–20). Springer-Verlag, Berlin, 2006, 81–94.
19. Filliâtre, J.-C. *Why: A Multi-Language Multi-Prover Verification Tool. Technical Report 1366*, Université Paris Sud, 2003.
20. Flanagan, C., Joshi, R., Ou, X., and Saxe, J.B. Theorem proving using lazy proof explication. In *Proceedings of the 15th International Conference on Computer Aided Verification* (Boulder, CO, July 8–12). Springer-Verlag, Berlin, 2003, 355–367.
21. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, June 17–19). ACM Press, New York, 2002, 234–245.
22. Ghilardi, S., Nicolini, E., and Zucchelli, D. A comprehensive framework for combined decision procedures. In *Proceedings of the Fifth International Workshop on Frontiers of Combining Systems, Vol. 3717 of LNCS*, B. Gramlich, Ed. (Vienna, Sept. 19–21). Springer-Verlag, Berlin, 2005, 1–30.
23. Godefroid, P., de Halleux, J., Nori, A.V., Rajamani, S.K., Schulte, W., Tillmann, N., and Levin, M.Y. Automating software testing using program analysis. *IEEE Software 25*, 5 (Sept./Oct. 2008), 30–37.
24. Grieskamp, W., Kicillof, N., MacDonald, D., Nandan, A., Stobie, K., and Wurden, F.L. Model-based quality assurance of Windows protocol documentation. In *Proceedings of the First International Conference on Software Testing, Verification, and Validation* (Lillehammer, Norway, Apr. 9–11). IEEE Computer Society Press, 2008, 502–506.
25. Henzinger, T.A., Jhala, R., Majumdar, R., and Sutre, G. Software verification with blast. In *Proceedings of the 10th International SPIN Workshop, Vol. 2648 of LNCS*, T. Ball and S. R. Rajamani, Eds. (Portland, May 9–10). Springer-Verlag, Berlin, 2003, 235–239.
26. Kaufmann, M., Manolios, P., and Moore, J.S. *Computer-Aided Reasoning: An Approach*. Kluwer Academic, June 2000.
27. Malik, S. and Zhang, L. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM 52*, 8 (Aug. 2009), 76–82.
28. McCarthy, J. Towards a mathematical science of computation. In Congress of the International Federation for Information Processing, 1962, 21–28.
29. Nelson, G. and Oppen, D.C. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1*, 2 (Oct. 1979), 245–257.
30. Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM 53*, 6 (Nov. 2006), 937–977.
31. Oppen, D.C. Complexity, convexity and combinations of theories. *Theoretical Computer Science 12*, 3 (1980), 291–302.
32. Owre, S., Rushby, J.M., and Shankar, N. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction* (Saratoga, NY, June 15–18). Springer-Verlag, Berlin, 1992, 748–752.
33. Ranise, S. and Tinelli, C. *The Satisfiability Modulo Theories Library (SMT-LIB)*, 2006; http://www.SMT-LIB.org
34. Tinelli, C. and Zarba, C.G. Combining nonstably infinite theories. *Journal of Automated Reasoning 34*, 3 (Apr. 2005), 209–238.

Leonardo de Moura (leonardo@microsoft.com) is a senior researcher in the Software Reliability Research group at Microsoft Research, Redmond, WA.

Nikolaj Bjørner (nbjorner@microsoft.com) is s senior researcher in the Foundations of Software Engineering group at Microsoft Research, Redmond, WA.