On Theorem Proving for Program Checking *

Historical perspective and recent developments

Maria Paola Bonacina

Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15, I-37134 Verona, Italy
mariapaola.bonacina@univr.it

Abstract

This article is a survey of recent results, related works and new challenges in automated theorem proving for program checking. The aim is to give some historical perspective, albeit necessarily incomplete, and highlight some of the turning points that made crucial advances possible.

Categories and Subject Descriptors I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—inference engines, resolution; D.2.4 [*Software Engineering*]: Software/Program Verification—formal methods, reliability, assertion checkers

General Terms Theory, Verification

Keywords Satisfiability modulo theories, Combination of theories, Rewrite-based theorem proving, Speculative inferences

1. Introduction

The design of computer programs that check whether other computer programs satisfy given properties is a central quest in computer science. On the one hand, software is so important, that we cannot abdicate from making it as reliable as possible; on the other hand, it is too complex to be checked manually, whence the grand challenge of reliable software, as outlined, for instance, in [34, 63, 86]. Much science and technology has been developed to improve *software reliability*, including:

- Testing (e.g., automated test case generation, automated or semi-automated testing),
- Static analysis (e.g., type systems, data-flow analysis, control-flow analysis, pointer analysis, symbolic execution, abstract interpretation),
- Dynamic analysis (e.g., traces, abstract interpretation),
- Software model checking (e.g., bounded model checking, counter-example guided abstraction refinement), and
- Deductive verification (e.g., weakest precondition calculi, verification condition generation and proof by theorem proving).

The complexity, variety, size and pervasiveness of software are such, that no approach is expected to be good enough on its own and in general. Rather, an increasingly wide-spread vision is to have a *pipeline of tools*, where technologies of increasing cost are applied to problems of increasing difficulty: for instance, first static analysis, then theorem proving, or first static analysis, then model checking with theorem proving.

A common trait of several of these approaches is some application of logic, ranging from type theories behind type systems, to temporal logics to write properties to be checked by model checking, to first-order theories to write annotations and verification conditions to be proved by theorem proving. If logic is used to formalize properties, then different sorts of mechanical reasoning, from model checking to theorem proving and model building, can be applied to reason about them. Historically, the notion that computation is a natural field of application for logic appeared as early as [81, 82], in the same epoch when *resolution* for mechanical theorem proving was invented [101]. A few years later it was discovered that program statements can be annotated with logical formulæ [52, 62]. At the same time, *paramodulation* and *demodulation* [100, 116], and *completion* with *superposition* and *simplification* by *rewriting* [69], began the study of inference rules that build the axioms of equality in the inference engine. Reasoning with equality and more generally *theory reasoning* are essential to reasoning about programs.

Since the early days, the history of research in program verification and in automated reasoning has continued, encountering both periods of successful synergy and others of detachment. The present time is one of renewed enthousiasm, due to recent progresses on both sides, revisitations of relatively older ideas in new light, and mergers of research streams that were separated for some time. One of the purposes of this article will be to highlight some of these connections. A current sentiment, shared in this article, is that not only deductive verification, that uses theorem proving to prove the validity of verification conditions, but pretty much all the verification technologies listed above can benefit from theorem proving (e.g., [10, 18, 59]).

2. Program checking and theorem proving

2.1 Software model checking

Perhaps a most well-known example is the usage of theorem proving in the *CounterExample-Guided Abstraction Refinement* (CEGAR) paradigm for software model checking (e.g., [2, 4, 11, 12, 33, 58, 60]). The essence of software model checking is to do model checking for infinite-state systems. Since the system has infinitely

* Supported in part by grant no. 2007-9E5KM8 of the Ministero dell'Istruzione Università e Ricerca.

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

many states, it is not possible to do model checking by enumerating execution paths, as in the original model checking, or by ultimately reducing the problem to propositional logic, as in symbolic model checking. Thus, one resorts to a combination of *abstraction*, *model checking* and *theorem proving*:

- In the abstraction phase, an *abstract program*, that represents an abstraction of the given program, is built automatically.
- In the model checking phase, a model checker is applied to check whether the abstract program satisfies the desired property. If it does, so does the program. Otherwise, an *abstract counterexample* is produced, together with a formula that is satisfiable if and only if the abstract counterexample is also a concrete counterexample.
- In the refinement phase, a theorem prover is applied to decide the satisfiability of this formula. If it is satisfiable, a program error has been found. Otherwise, it is not necessarily the case that the program is correct, because the abstraction may have been too coarse. Thus, information extracted from the proof of unsatisfiability is applied to refine the abstraction before repeating the cycle.

A notion of *abstraction* in first-order theorem proving appeared in [96], where abstraction is generalization, such as replacing terms by new variables, to capture the idea that a more general theorem may be easier to prove than the original one. An abstraction from first-order to propositional logic maps first-order atoms to propositional variables. When this sort of abstraction is applied to an annotated program, the abstract program is a *boolean program* (e.g., [60]). If abstraction is restriction to linear arithmetic, the abstract program is a *linear program*, where the allowed atoms involve linear expressions on integer or real variables (e.g., [4]).

A refinement of the abstraction means allowing a richer language, with more predicates, hence more atoms. The proof of unsatisfiability is used to determine which predicates to add. This choice is crucial to limit the amount of repetition between one round of the CEGAR paradigm and the next. The goal is to add a minimal set of predicates, while making sure that the abstract counterexample that was already considered will not be considered again. Then, the prover needs to produce a minimal unsatisfiable set of literals, called *unsatisfiable core*, because it represents the core of the proof of unsatisfiability.

2.2 Interpolation

In this context, a counterexample is a representation of an execution trace, such as a sequence of program locations that leads to an error location. Many or even infinitely many concrete traces correspond to an abstract trace, and even at the abstract level different traces have subtraces in common. The refinement ought to be so precise to detect which predicates to add to exclude specific subtraces or even states. Decomposing traces into subtraces involves finding intermediate states. Since formulæ are associated to program states, finding intermediate states corresponds to finding intermediate formulæ, or formulæ that lie between given formulæ.

This intuition is expressed formally by the notion of *interpolant*: given formulæ A and B, an interpolant of (A, B) is a formula I, such that A entails I, I entails B, and I is built only out of symbols common to A and B. The trace from A to B can be decomposed into subtraces from A to I and from I to B. If I is a *reverse interpolant* of (A, B), that is, an interpolant of $(A, \neg B)$, the addition of literals from I to the abstraction may exclude the subtrace from I to B, which may be erroneous or spurious, while saving that from A to I. More generally, I can be seen as an *approximation* of A directed by the goal of reaching, or excluding, B. Thus, theorem provers are used to generate interpolants (e.g., [61, 66, 67, 72, 84, 117]).

2.3 Deductive verification

The rôle of theorem proving in deductive verification is even more direct (e.g., [13, 29, 51, 75, 76, 79, 80]):

- In the annotation phase, the given program is annotated with *assertions*, including *function specifications*, that is preconditions and postconditions for each function, *loop invariants*, *runtime assertions* and *function call assertions*. *Program variables* appear in assertions as *free variables*. A *program state* is an assignment to program variables, hence to free variables, which can be extended to an interpretation, so that it makes sense to check whether it satisfies an assertion.
- In the generation phase, a *verifying compiler* decomposes the annotated program into *basic paths*. For each basic path, it propagates the given postcondition backward to compute a *weakest precondition*. Then, the *verification condition* is that the given precondition implies this weakest precondition. If the verification conditions are valid, the given assertions are *inductive invariants*, or *invariants* for short, of the program, which means that the program satisfies its specification.
- In the validation phase, a theorem prover is applied to determine whether the verification conditions are valid. If the answer is positive, the prover produces a proof of validity, usually a proof of unsatisfiability of the negation of the verification condition, since the reasoning is refutational. Otherwise, it produces a model of the negation of the verification condition, that is a counter-model of the verification condition itself. This counter-model is a counterexample to the notion that the assertions are inductive invariants, and therefore can be used to detect an error, either in the code or in the assertions.

A quantifier-free formula with free variables is valid if and only if its universal closure is. In refutational theorem proving, through negation and skolemization, a universally quantified variable is replaced by a constant. Thus, program variables become constants from a theorem-proving point of view. This is a reason why it happens that symbols that are called "variables" in the verification literature are called "constants" in the theorem proving literature: in the former, one is mindful of the origin of the symbols, whereas in the latter, one is careful about the presence of quantifiers, since it has an impact on the difficulty of the problem and on the inference engine to be used.

2.4 Invariant generation

Some annotations, such as runtime assertions (e.g., bounds on array indices), can be generated automatically by the verifying compiler itself. However, this is not true in general for all annotations, and since annotating programs by hand is cumbersome and expensive, manual annotation is an obstacle to the wide-spread acceptance of these technologies [76]. Thus, there is much interest in automating the annotation process, or, better still, in generating automatically invariants, that is, valid annotations. Here too, theorem proving can be of help: a theorem prover can be seen not only as a *proof procedure*, that searches for a proof of a submitted conjecture, but also as a *completion procedure*, that completes a given set of formulæ into a *complete* or *saturated* set.

This notion appeared as early as [69], for the special case of *confluent rewrite systems*, and has been developed and generalized by many authors, including [6–8, 19, 20, 24, 47, 64, 71]. A set of formulæ is a *presentation* of a theory. Operationally, a set is saturated if all inferences are redundant. If inferences are conceived as normalizing proofs by transforming presentations, then a complete

presentation is one which affords a normal form proof for every theorem, and a saturated presentation is one which provides all the normal form proofs of all theorems.

Invariant generation can be approched by submitting to a theorem prover an appropriate set of formulæ about the program, and extracting invariants during the saturation, even if the saturation may not halt. The crux is to design which formulæ to submit, and which ones to filter from the output. Furthermore, the extracted formulæ may need to be post-processed: for instance, they may be deskolemized to put back existential quantifiers in place of Skolem terms. This line of research was explored already in [31], and recently was investigated in [73, 85].

2.5 Static analysis

More effort has been devoted to automate invariant generation by *static analysis* (e.g., [29, 112]). The goal is to generate an assertion for the beginning and ending locations of all basic paths. The key idea is that of *symbolic execution*, where a formula is propagated through the program over basic paths. In *forward propagation*, a precondition is propagated forward by computing *strongest postconditions*. In *backward propagation*, a postcondition is propagated backward by computing *weakest preconditions*. Either process requires to determine the validity of implications: for example, forward propagation requires to determine whether the strongest postcondition computed over a path implies the current postcondition of the path; if not, the current postcondition needs to be updated. Thus, a static analyzer performing a symbolic execution invokes a theorem prover to decide the validity of implications. Clearly, invariant generation by a static analyzer and invariant checking by a verifying compiler are complementary and may be combined. Both call upon a theorem prover to determine the validity of formulæ.

In a real execution, a program is applied to a specific input, and it is in a unique state at every stage of the execution. In a symbolic execution, a program is applied to formulæ: since a formula represents all the states that satisfy it, the program is in a set of states at every stage of a symbolic execution. In principle, the symbolic execution tries to generate for each location a formula so precise, that it is satisfied by all and only the states that the program can be in, when its control reaches that location. The price to pay for such an ideal precision is that the symbolic execution does not terminate even on very simple programs. In order to enforce termination, one resorts to *abstraction*. In static analysis with *abstract interpretation* (e.g., [34, 35, 68, 104]), the realm of admissible formulæ is restricted to an *abstract domain*, that is, a syntactically defined class of formulæ, that can state only certain properties (e.g., numerical constraints on program variables). As in [96], and in the CEGAR paradigm, the abstraction involves a map, called *abstraction function*, from the language of first-order formulæ to a simpler language. All the formulæ that cannot be expressed in the target language are mapped to $\top$, or $true$, and therefore ignored, whence the abstraction. The trade-off between *termination* and *scalability*, on the one hand, and *expressivity* and *precision*, on the other, is at the heart of this research.

3. Decision procedures

In all the above contexts, it is crucial that the theorem prover is a *decision procedure* for validity, so that it is guaranteed to terminate. In first-order logic, validity, or, equivalently, unsatisfiability, is only semi-decidable. Its complement, invalidity, or, equivalently, satisfiability, is not even semi-decidable. Thus, it is necessary to restrict the attention to decidable theories or fragments. If validity is decidable, then also satisfiability is decidable. This is an additional reason why both software model checking and static analysis resort to *abstraction*: the less expressive language of the abstract domain should have a decidable validity problem. Certain abstractions are so strong, that they yield domains where implication queries can be answered directly by the static analyzer without even the need for a decision procedure: this is the case, for instance, for interval analysis or Karr's analysis [29, 35, 68]. Clearly, such domains have limited expressive power.

3.1 Propositional satisfiability

Aside from those abstractions that do not even require a decision procedure, a most basic abstraction is the one that maps first-order formulæ to propositional formulæ. Regardless of abstraction, a decision procedure for propositional logic is anyway essential to handle boolean connectives. Among the many decision procedures for propositional logic, the *Davis-Putnam-Logemann-Loveland (DPLL) procedure* emerged as a standard choice in theorem proving for program checking. Originally conceived for first-order logic [37, 38], its basic mechanism consists of searching for a model, by *deciding*, or *guessing*, the truth value of each propositional variable, and then propagate it (*boolean constraint propagation* (BCP)) over the given set of propositional formulæ, typically clauses. The procedure maintains *a set F of clauses* to be satisfied and *a current assignment M*, that represents a candidate model. Since there are finitely many variables, the search is done by *depth-first search*. When the assignment falsifies a clause (*conflict clause*), the procedure undoes decisions by backtracking. If it finds a model, it returns satisfiable, and unsatisfiable, otherwise.

From a theorem proving point of view, its distinctive advantage is that the assignment of truth values to propositional variables has the effect of *splitting* disjunctions (e.g., given $A \vee B$, consider first the case where A is true, and if that fails, the case where A is false and B needs to be true). This breaking apart of disjunctions is very important in theorem proving for program checking, because verifying compilers and static analyzers tend to generate formulæ with huge ground disjunctions (e.g., with a disjunct for every possible number of iterations of a loop).

The success of DPLL was made possible by an impressive history of advances in its understanding and implementation (e.g., [40, 48, 66, 87, 118, 119]), including:

- Choice of normal form: *negation normal form* (NNF), with the so-called *Tseitin encoding* [113], yields *equisatisfiable conjunctive normal form* (ECNF), which avoids the duplication of subformulæ by distributivity of plain conjunctive normal form (CNF);
- Data structures and algorithms for fast BCP: the *two watched literals scheme* reckons that a clause is neither a conflict clause (all literals false) nor it contains an *implied literal* (all literals false except one), as long as it contains two non-false literals, so that it is sufficient to watch two literals at a time in every clause;
- *Conflict-driven backjumping*: the conflict clause yields a series of *explanation* steps by resolution, that generate other conflict clauses, until an *asserting clause C*, that is, a conflict clause with only one literal l assigned in the current decision level, is generated; then C is added to F, the value of l is flipped, and the procedure backjumps to the earliest decision level where l is unassigned and all other literals of C are false, so that l is implied by C and C is satisfied.

In the following, a *SAT-solver* is a theorem prover implementing DPLL along these lines.

3.2 Satisfiability modulo theories

Assertions about programs need to state properties about numbers, and about data structures, such as lists, queues, arrays, records, sets, multisets, hashtables, in their various flavours (e.g., singly-

linked lists and doubly-linked lists). From a theorem proving point of view, these are first-order theories defined by a signature Σ, that includes equality, and a presentation $\mathcal{T}$. The most common and most studied of these theories include *equality*, *linear arithmetic*, *recursive data structures* with *constructors* and *selectors*, *lists*, *arrays* and *bitvectors*. For the theory of equality, $\mathcal{T}$ contains only the axioms of equality. For this reason, this theory is also called *equality with uninterpreted function symbols* (EUF): equality is the only predicate symbol and all other symbols are *uninterpreted* function symbols, since there are no axioms restricting their interpretation. Uninterpreted predicates can be replaced by uninterpreted functions. Linear arithmetic will not be covered here: the interested reader may read, for instance, [29, 50, 78]. Similarly, references for the theory of bitvectors include [30, 36, 44]. For lists there are different axiomatizations, including: non-empty and possibly cyclic lists [3, 89, 108], possibly empty and possibly cyclic lists [5], non-empty and acyclic lists. The latter is the instance with one constructor and two selectors of the theory of recursive data structures [21, 94]. For arrays, one distinguishes between the theory of *arrays without extensionality*, which can express equality between elements or between indices of arrays, and the theory of *arrays with extensionality*, which can express also equality between arrays (e.g., [3, 5, 44, 110]).

An interpretation that satisfies $\mathcal{T}$ is a $\mathcal{T}$-model. *$\mathcal{T}$-satisfiability*, or *satisfiability modulo $\mathcal{T}$* (SMT), is the problem of determining whether a $\mathcal{T}$-formula is $\mathcal{T}$-satisfiable, or has a $\mathcal{T}$-model. For most first-order theories, $\mathcal{T}$-satisfiability is only semi-decidable, but it is decidable in the quantifier-free fragment of several theories. A decision procedure for $\mathcal{T}$-satisfiability in the quantifier-free fragment of $\mathcal{T}$ is usually called a *$\mathcal{T}$-decision procedure*. Since a quantifier-free formula can be reduced to a conjunction of ground $\mathcal{T}$-clauses, the input for a $\mathcal{T}$-decision procedure is typically a set of ground $\mathcal{T}$-clauses. A simpler instance of the problem is to decide the $\mathcal{T}$-satisfiability of *a set S of ground $\mathcal{T}$-literals*. A decision procedure for $\mathcal{T}$-satisfiability of sets of ground $\mathcal{T}$-literals is usually called a *$\mathcal{T}$-satisfiability procedure*. In principle, a $\mathcal{T}$-satisfiability procedure would suffice, because a quantifier-free formula can be reduced to *disjunctive normal form* (DNF): if all disjuncts are found unsatisfiable, the formula is unsatisfiable, and satisfiable otherwise. This reduction is not considered practical in general, because of the duplication of subformulæ caused by distributivity. However, problems reduce to sets of literals through negation and skolemization: for instance, $\mathcal{T}$-validity of a $\mathcal{T}$-clause $\forall \bar{x}\ C$ is equivalent to $\mathcal{T}$-unsatisfiability of a set of ground $\mathcal{T}$-literals. Also, the assignment M in DPLL is a set of literals, and $\mathcal{T}$-decision procedures are obtained by integrating $\mathcal{T}$-satisfiability procedures in DPLL.

If $\mathcal{T}$ is the theory of equality, the quantifier-free fragment is decidable, and S is a set of ground equalities and negated equalities. The algorithm of choice for this problem is the *congruence closure* (CC) algorithm (e.g., [48, 49, 89, 107]). This algorithm reasons about equality in a bottom-up fashion, deducing that two terms with the same top function symbol are congruent, if their arguments are pairwise congruent. All terms in S are represented in a directed acyclic graph, called *E-graph*, where the consequences of every discovered congruence propagate. The congruence closure operations are implemented on the graph by *union* (to unite congruence classes) and *find* (to produce the representative of each class) steps. If two terms sides of a negated equality turn out to be congruent, the algorithm returns unsatisfiable, and satisfiable otherwise.

For a set of ground literals, completion with superposition and rewriting reduces to *ground completion*, where superposition reduces to rewriting, since there is no need for unification. That ground completion can compute congruence closure was known since [77]. Symmetrically, congruence closure can be used to compute ground completion [32, 54, 109]. The analysis in [9] showed that in terms of algorithmic complexity congruence closure and ground completion are comparable. What made the fortune of congruence closure is that it proved to be a building block for:

- $\mathcal{T}$-satisfiability procedures for theories other than equality, obtained by *building* the axioms of $\mathcal{T}$ *into* the CC algorithm, as shown already in [89], for the theory of non-empty and possibly cyclic lists, and later, for instance, in [44, 110], for those of arrays with or without extensionality: key moves are adding to the E-graph the terms of the instances of axioms of $\mathcal{T}$, obtained by replacing their universally quantified variables with terms in S, viewing the E-graph as central repository for both equality and theory reasoning, and applying the axioms taking into account the congruence being built (e.g., [44, 48]);
- $\mathcal{T}$-satisfiability procedures for *combinations of theories* $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$, based on the *equality sharing* method and its variants (e.g., [29, 88, 95, 102, 108, 111]); and
- $\mathcal{T}$-decision procedures obtained by integrating DPLL, to handle the boolean structure of the formula, with $\mathcal{T}$-satisfiability procedures, to do the theory reasoning, according to the *DPLL($\mathcal{T}$) paradigm* (e.g., [93, 106]).

The equality sharing method and the DPLL($\mathcal{T}$) paradigm are covered in the next two subsections.

3.3 Equality sharing

The problem of *combination of theories* is to obtain a $\mathcal{T}$-satisfiability procedure for $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$, given $\mathcal{T}_i$-satisfiability procedures, for $1 \leq i \leq n$. For simplicity, let $n = 2$. Combination of theories is a fundamental issue in theorem proving for program checking, since the queries generated by verifying compilers or static analyzers typically involve multiple theories (e.g., integers and arrays, integers and lists, arrays and bitvectors). Intuitively, it is desirable to minimize communication: why should a procedure for the theory of arrays be concerned about arithmetic or vice versa? This is even more true for procedures that build the axioms of the theory into the algorithm, precisely because knowledge about the theory is embedded in the algorithm. To begin with, the *equality sharing method* requires that the $\mathcal{T}_i$'s are *disjoint*, which means that they do not share function or predicate symbols other than equality. However, the terms in S typically do mix symbols from different theories. Then, the first step is to *separate* occurrences of symbols from different theories, by introducing *new constant symbols*. For example, $f(g(a)) \simeq b$, where f and g belong to the signatures of different theories, becomes $f(c) \simeq b \wedge g(a) \simeq c$, where c is new. S is partitioned into two sets S_1 and S_2 such that S_1 is a set of $\mathcal{T}_1$-literals, S_2 is a set of $\mathcal{T}_2$-literals, and they share only constants: since only constants are introduced, $S_1 \cup S_2$ remains ground, and since the constants are new, $S_1 \cup S_2$ and S are $\mathcal{T}$-equisatisfiable.

Let V be the set of shared constants. In order to decide $\mathcal{T}$-satisfiability of $S_1 \cup S_2$ it is sufficient to guess an *arrangement* of V, that is, whether $a \simeq b$ or $a \not\simeq b$ for every pair $a, b \in V$, and let the procedures for $\mathcal{T}_1$ and $\mathcal{T}_2$ check whether the arrangement is acceptable for the respective theories. Formally, $S_1 \cup S_2$ is $\mathcal{T}$-satisfiable if and only if there exists an arrangement K such that $S_1 \cup K$ is $\mathcal{T}_1$-satisfiable and $S_2 \cup K$ is $\mathcal{T}_2$-satisfiable. In practice, each procedure deduces the equalities between shared constants entailed by its $\mathcal{T}_i \cup S_i$ and propagates them to the other. This is sufficient for completeness if each $\mathcal{T}_i$ is *convex*, that is, whenever $\mathcal{T}_i \models H \Rightarrow \bigvee_{j=1}^{m} a_j \simeq b_j$, where H is a conjunction of atoms, then $\mathcal{T}_i \models H \Rightarrow a_j \simeq b_j$, for some j, $1 \leq j \leq m$. Convexity excludes the situation where all models satisfy some disjunct, but no disjunct is satisfied by all models. Clearly, if all theories are convex, it is not necessary to reason about disjunctions. For instance, Horn theories are convex.

If a theory is not convex, its procedure needs to propagate all entailed *disjunctions of equalities* between shared constants. This generalization is sufficient for completeness, if each theory $\mathcal{T}_i$ is *stably infinite*, that is, every $\mathcal{T}_i$-satisfiable ground formula has a $\mathcal{T}_i$-model with domain of infinite cardinality. For first-order theories with no trivial models, convexity implies stable infiniteness [17, 55]. The meaning of stable infiniteness is less intuitive than that of convexity: technically, what needs to be exchanged between the theories are *interpolants*, whence the emphasis on shared symbols, and stable infiniteness is needed to make sure that it is sufficient to propagate quantifier-free interpolants. More precisely, the infinite supply of elements of the infinite domain guarantees *quantifier elimination*, or that an interpolant with quantifiers can be replaced by an equivalent interpolant without quantifiers (e.g., [29, 55]).

In the following, a *$\mathcal{T}$-solver* is an engine implementing a $\mathcal{T}$-satisfiability procedure for a combination $\mathcal{T}$ of theories.

3.4 SMT-solvers

Many problems in computer science can be reduced to instances of propositional satisfiability (SAT). This tradition and the amazing progress in the efficiency of SAT-solvers made popular for some time the notion of attacking $\mathcal{T}$-decision problems by reducing them to instances of SAT and applying a SAT-solver. This approach was called *eager* for the eagerness of the reduction to SAT. Combination of theories is not an issue, since all get reduced to propositional logic. Two drawbacks of reduction to SAT are the loss of problem structure and the space complexity of the reduction, which relates the size of the resulting formula to that of the original one. Even a quadratic reduction, which may sound efficient in theory, may be problematic in practice, since the size of relevant formulæ is of the order of megabytes. Although there are specific theories or classes of problems for which reduction to SAT may be the best option, the notion that it may represent a general solution for all $\mathcal{T}$-decision problems for all theories has been abandoned. The expressivity requirements of problems generated by program checking certainly played a rôle in this evolution.

The *DPLL($\mathcal{T}$) paradigm* integrates the SAT-solver with a $\mathcal{T}$-solver, in such a way that the SAT-solver searches for a model of the formula, and the $\mathcal{T}$-solver ensures that the propositional model is also a $\mathcal{T}$-model. In the first trials (e.g., [16, 45]), the SAT-solver would generate a complete propositional model and then call the $\mathcal{T}$-solver to check it. For this reason, and to differentiate it from the eager approach, this style was called *lazy*. However, such a loose integration cannot be sufficiently efficient, because of the work wasted by the SAT-solver pursuing candidate models that are then discarded by the $\mathcal{T}$-solver. Thus, the prevailing approach, still called "lazy" or sometimes "hybrid" or simply DPLL($\mathcal{T}$), is a tight integration, where the SAT-solver propagates to the $\mathcal{T}$-solver every truth assignment. The $\mathcal{T}$-solver responds by signalling whenever a subset of the current assignment M is $\mathcal{T}$-inconsistent (*$\mathcal{T}$-conflict*), and by propagating to the SAT-solver ground literals that M entails in $\mathcal{T}$ (*$\mathcal{T}$-propagate*). In other words, the notions of *conflict* and *implied literal* of DPLL are generalized to conflict and implication modulo $\mathcal{T}$.

The CC algorithm and the $\mathcal{T}$-solver reason in first-order logic, while the SAT-solver reasons in propositional logic, so that the interface between them involves an *abstraction function* mapping first-order atoms to propositional variables and vice versa. Propositional variables standing for first-order atoms are sometimes termed *proxy variables*. Theorem provers implementing this paradigm are called *SMT-solvers* (e.g., CVC and its successor CVC Lite [15], Simplify [48], MathSAT [27], ICS [46] and its successor Yices [50], Barcelogic [91], Z3 [43]).

The development of DPLL($\mathcal{T}$)-based SMT-solvers affected the implementation of equality sharing:

- The propagation of disjunctions required by non-convex theories is realized by case analysis and backtracking, that is, the $\mathcal{T}$-solver propagates one literal of the disjunction, and if the guess fails, it will be undone by backtracking, and another literal of the disjunction will be tried. A disjunction due to non-convexity is simply another clause to satisfy.
- In *delayed theory combination* (DTC) [28] the SAT-solver interacts with n $\mathcal{T}_i$-solvers, rather than with one $\mathcal{T}$-solver for $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$. The SAT-solver is endowed from the start with proxy variables for all possible equalities between shared constants, and it computes their arrangement by guessing assignments and propagating them to the $\mathcal{T}_i$-solvers that check them for $\mathcal{T}_i$-consistency. The *plunging on the literal* technique of [48] could be seen as an ancestor of DTC. Since DTC may yield too much trial and error, many variations have been studied (e.g., [14, 74]).
- *Model-based theory combination* [42] requires that each $\mathcal{T}_i$-solver maintains a candidate $\mathcal{T}_i$-model consistent with the current assignment M. Each $\mathcal{T}_i$-solver is allowed to propagate equalities between shared constants, that are true in its candidate model, regardless of whether they are entailed: if they generate conflicts, they will be undone by backtracking, and the $\mathcal{T}_i$-solver will update its $\mathcal{T}_i$-model accordingly. Since it is generally less expensive to produce the equalities satisfied by a specific $\mathcal{T}_i$-model than those satisfied by all $\mathcal{T}_i$-models consistent with M, and the number of equalities that matter in practice is small, it pays off to be optimistic and try those that are easier to generate.

A significant body of work has been spent to extend equality sharing to non-disjoint combinations or beyond stably infinite theories. Theoretical investigations include [53, 57, 90, 115]. For example, the theory of fixed-size bitvectors is not stably infinite. The approach of [44] is based on the observation that this theory is *strongly disjoint* from other theories, such as linear arithmetic, meaning that not only there are no shared function or predicate symbols, but also no shared sorts. A few strongly disjoint theories, namely bitvectors, linear arithmetic, scalar values and boolean values, are combined with EUF by model-based combination, to form a *core theory*. Then, other theories, such as arrays, are reduced to the core theory by a model-based reduction.

From now on, an *SMT-solver* is an engine implementing DPLL($\mathcal{T}$) for a combination $\mathcal{T}$ of theories.

4. General theorem proving

The proof obligations that verifying compilers or static analyzers generate for theorem provers are not restricted to quantifier-free problems. Quantifiers are needed to write, for instance, frame conditions over loops, invariants about arrays or heaps, and to axiomatize theories, such as type systems, for which decision procedures for ground formulæ are not available. A typical verification problem consists of determining the satisfiability modulo $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$ of a set of formulæ $\mathcal{R} \cup P$, where $\mathcal{R}$ contains non-ground clauses, and P is a large ground formula, or set of ground clauses. Usually, $\mathcal{T}$-symbols occur in P, but not in $\mathcal{R}$, which can be regarded as the axiomatization of an application specific theory.

4.1 E-matching

Most SMT-solvers are restricted to ground formulæ, and may instantiate quantified variables only by heuristic techniques, known as *E-matching* [39, 48, 56]. Let E be the set of equations currently represented in the E-graph, and let $t[x]$ be a term in a non-ground clause C with a universally quantified variable x. The idea of E-matching is to instantiate C with those ground substitutions σ such

that $t[x]\sigma \simeq_E s$, where s is a ground term in the E-graph, and $\simeq_E$ is equality modulo E. Since there are finitely many ground terms in the E-graph, only finitely many ground instances will be considered, saving termination at the expense of completeness. For reasons of efficiency, not all non-ground terms $t[x]$ are necessarily considered. The procedure is usually restricted to those non-ground terms $t[x]$ selected by the user with an appropriate syntax in every non-ground clause C. These selected terms are called *triggers*, because they "trigger" the instantiation mechanism.

An advantage of E-matching is that it takes into account the information in the E-graph. This seems to be a reason why E-matching is efficient, when it works. On the other hand, the choice of triggers is not only problem-dependent, but also prover-dependent; and it may take weeks to the user to guess the right triggers. The incompleteness due to this heuristic handling of quantifiers may cause false positives: the software model checker will conclude that an abstract counterexample is a concrete counterexample when it is not; the verifying compiler will conclude that a verification condition is not valid, when it is; in both cases spurious errors will be reported. For a static analyzer generating invariants, an erroneous rejection of valid conjectures will cause the production of less precise invariants.

4.2 Rewrite-based decision procedures

A natural choice to reason in a complete way about quantifiers is to resort to a theorem prover for first-order logic with equality (e.g., Otter [83], E [105], SPASS [114], Vampire [99], to mention a few with a long history of development and application). Several state-of-the-art theorem provers for first-order logic with equality implement inference systems issued from the merger of resolution and completion, called at various points in time *resolution-based*, *rewrite-based*, *completion-based*, *superposition-based*, *paramodulation-based*, *contraction-based*, *saturation-based* or *ordering-based*, to emphasize one aspect or the other (e.g., [7, 19, 24, 65, 92, 103]). Advantages of generic first-order systems include: *expressivity*, *soundness* and *completeness*, powerful *contraction* rules to remove redundant formulæ, *proof generation*, *model generation* from finite saturated sets, combination of theories by taking as input the union of their presentations, and theory-independent support of all these features. The crux is termination. However, a series of somewhat surprising results showed that first-order theorem proving strategies can be decision procedures for satisfiability modulo theories.

Consider a rewrite-based strategy whose only built-in theory is equality, so that $\mathcal{T}$ may stand for $\mathcal{T} \cup \mathcal{R}$. Such a strategy is guaranteed to terminate on $\mathcal{T}$-satisfiability problems $\mathcal{T} \cup S$, in several theories of interest for program checking, including *equality*, *lists*, *recursive data structures*, *arrays*, *finite sets* and *records*, all three with or without extensionality, and two fragments of arithmetic, *integer offsets*, and *integer offsets modulo*, used with arrays to model *queues* and *circular queues* [3, 5, 21, 22]. The proofs of termination are obtained by analyzing the inferences from $\mathcal{T} \cup S$, and showing that only finitely many clauses can be generated. Thus, a theorem-proving strategy is a $\mathcal{T}$-satisfiability procedure. The experiments in [5] compared the rewrite-based theorem prover E 0.82 with the SMT-solvers CVC 1.0a and CVC Lite 1.1.0, with results that were comparable or even favorable to the prover. Shortly after, it was proved that a rewrite-based strategy is a *polynomial* $\mathcal{T}$-satisfiability procedures for *records with extensionality* and *integer offsets* [22]. These findings dispelled the folklore that the only way to reason effectively about such theories would be to build their axioms in the CC algorithm.

4.3 Variable inactivity

Combination of theories is approached by a *modularity theorem* stating sufficient conditions for termination on a union of theories, given termination on each [5]. The sufficient conditions are that the theories are *disjoint* and *variable-inactive*: the latter property means that no persistent irredundant clause generated by a fair strategy from $\mathcal{T} \cup S$ has a maximal literal $t \simeq x$ with $x \notin Var(t)$. All the above mentioned theories satisfy these conditions, and therefore a fair rewrite-based strategy is a $\mathcal{T}$-satisfiability procedure for any of their combinations. Intuitively, disjointness and variable inactivity prevent unbounded inferences across theories, so that termination on each theory is sufficient to get termination in the union. Disjointness prevents paramodulations from compound terms, and variable inactivity prevents paramodulations from variables, so that the only inferences across theories are paramodulations from constants, that are bounded by the number of constant symbols.

This result is an analogue of equality sharing for general theorem proving, where paramodulations from constants correspond to the propagation of equalities between constants. Indeed, *variable inactivity implies stable infiniteness* [5, 25]: if a theory is not stably infinite, a fair rewrite-based strategy is guaranteed to generate eventually a *cardinality constraint* (e.g., $y \simeq x \vee y \simeq z$), which is not variable-inactive. Thus, a rewrite-based strategy can be used to discover the lack of stable infiniteness. Furthermore, if $\mathcal{T}$ is variable-inactive, and a rewrite-based strategy is a $\mathcal{T}$-satisfiability procedure, then it is also a *$\mathcal{T}$-decision procedure* [22]. The proof is based on an analysis of inferences in a variable-inactive theory.

4.4 Decision procedures by stages

The direct application of a first-order theorem prover to a verification problem $\mathcal{T} \cup \mathcal{R} \cup P$ is not expected to work well in practice, for at least two reasons:

- Resolution recombines in the resolvent most of the literals of its parent clauses. Because of this *duplication by combination* [97], resolution is not designed for propositional efficiency, especially on non-Horn clauses. Its strength is the use of unification to instantiate universally quantified variables at the first-order level, rather than the mere recombination of literals at the ground level. Since P typically contains huge non-Horn clauses, it is preferable to handle them by the case analysis by splitting of DPLL.
- Theories such as *linear arithmetic* and *bitvectors* require computing and solving rather than deducing, and these theories appear very often in the combination $\mathcal{T}$ and in the ground part P of the problem.

First-order theorem provers are strong at reasoning with non-ground first-order clauses and ground unit first-order clauses, while SMT-solvers are strong at reasoning with propositional clauses and embedding special theories. Since they complement each other, approaches to integrate them are being investigated. In *decision procedures by stages* [23], a rewrite-based strategy is applied in a first stage to do theory reasoning in $\mathcal{R}$, and an SMT-solver is applied in a second stage to do propositional reasoning and theory reasoning in the built-in theory $\mathcal{T}$. The ground set P is decomposed into $P_1 \cup P_2$ in such a way that P_1 contains unit $\mathcal{R}$-clauses and P_2 contains everything else. In the first stage, $\mathcal{R} \cup P_1$ is completed in a saturated set $\mathcal{R} \cup \bar{P}$. In the second stage, $\bar{P} \cup P_2$ is passed to an SMT-solver that decides its satisfiability modulo $\mathcal{T}$. Under suitable hypotheses, $\bar{P} \cup P_2$ is finite, and one obtains an *$\mathcal{R}$-decision procedure* by an inference-based, rather than model-based, reduction of $\mathcal{R}$ to $\mathcal{T}$. The theories of *arrays* and *records*, both with or without extensionality, *integer offsets*, and their combinations, satisfy the hypotheses and get reduced to the theory of equality. Decision by

stages is *modular* with respect to combination of theories, since different theories can be completed independently, and it allows one to leave in $\mathcal{T}$ theories such as linear arithmetic and bitvectors.

4.5 DPLL($\Gamma + \mathcal{T}$) and speculative inferences

In decision by stages the rewrite-based strategy is used as a completion procedure to generate a saturated set. A tighter integration of theorem prover and SMT-solver is obtained by conceiving the theorem prover as a generic satellite solver for any $\mathcal{R}$ for which the SMT-solver does not have a built-in decision procedure. This is the idea of the *DPLL($\Gamma + \mathcal{T}$) inference engine* [26, 41], where Γ is a generic inference system based on resolution, superposition and rewriting. In order to maximize their synergy, DPLL($\mathcal{T}$) works on ground clauses and literals, whereas Γ works on non-ground clauses and unit ground $\mathcal{R}$-clauses. The integration of Γ within DPLL($\mathcal{T}$) affects several aspects:

- *Deduction*: Γ-inferences may take as premises clauses in F and $\mathcal{R}$-literals in M, and add to F the clauses thus generated; clauses are replaced by hypothetical clauses $H \triangleright C$, where C is a clause, the hypothesis H is the set of ground $\mathcal{R}$-literals from M that C depends on, and $(l_1 \wedge \ldots \wedge l_n) \triangleright (l'_1 \vee \ldots l'_m)$ is interpreted as $\neg l_1 \vee \ldots \vee \neg l_n \vee l'_1 \vee \ldots \vee l'_m$; Γ-inferences essentially ignore the hypotheses of their premises, except that conclusion inherits them as hypotheses together with any $\mathcal{R}$-literals from M used as premises.
- *Backjumping*: hypothetical clauses depending on literals retracted by backjumping are removed from F.
- *Contraction*: while contraction rules that simply delete a clause, such as *tautology deletion*, apply to $H \triangleright C$ like to C, contraction rules that justify the removal of $H \triangleright C$ by other clauses, such as *subsumption* and *simplification*, are modified to take into account dependence on M: for instance, assume that D subsumes C, so that $H_2 \triangleright D$ subsumes $H_1 \triangleright C$; let $level(H)$ be the maximum among the decision levels the literals of H belong to; $H_1 \triangleright C$ is deleted only if $level(H_1) \geq level(H_2)$; otherwise, $H_1 \triangleright C$ is only disabled and will be re-enabled when $level(H_2)$ is backjumped. The condition $level(H_1) \geq level(H_2)$ ensures that $H_1 \triangleright C$ would be removed upon backjumping *before* $H_2 \triangleright D$; it prevents the unsound situation where $H_1 \triangleright C$ is subsumed when the clause that subsumes it is gone because of backjumping.

DPLL($\Gamma + \mathcal{T}$) uses equality sharing implemented by model-based theory combination to combine the built-in theories in $\mathcal{T}$ and variable inactivity to combine the axiomatized theories in $\mathcal{R}$ [26]. Thus, its refutational completeness requires that $\mathcal{T}_1, \ldots, \mathcal{T}_n$ and $\mathcal{R}$ are pairwise *disjoint*, $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are *stably infinite*, and $\mathcal{R}$ is *variable-inactive*. Since variable inactivity implies stable infiniteness, it allows the system to combine built-in and axiomatized theories, and to detect unsatisfiability due to the lack of infinite models, if Γ generates a cardinality constraint. To obtain a complete strategy, a refutationally complete inference system needs to be coupled with a fair search plan. In the presence of first-order clauses and first-order inferences, the search space is not finite, and the depth-first search plan of DPLL($\mathcal{T}$) is not fair. Thus, DPLL($\Gamma + \mathcal{T}$) resorts to *depth-first search with iterative deepening* on the depth of inferences.

Up to here, DPLL($\Gamma + \mathcal{T}$) is a semi-decision procedure for validity. In order to get decision procedures, it is equipped with the capability of performing *speculative inferences* [26]. The intuition comes from the observation that axioms such as *transitivity* ($\neg(x \sqsubseteq y) \vee \neg(y \sqsubseteq z) \vee x \sqsubseteq z$) and *monotonicity* ($\neg(x \sqsubseteq y) \vee f(x) \sqsubseteq f(y)$), that arise in formalizations of type systems, where $\sqsubseteq$ is a subtype relationship, and f a type constructor such as *Array-of*, are problematic, because they generate an unbounded number of clauses. For example, resolution, even with negative selection [7], would generate an infinite series $\{f^i(a) \sqsubseteq f^i(b)\}_{i \geq 0}$ from monotonicity and a literal $a \sqsubseteq b$. In practice, it is seldom the case that one needs to go beyond $f(a) \sqsubseteq f(b)$ or $f^2(a) \sqsubseteq f^2(b)$ to decide satisfiability. The idea is to allow the prover, or the experimenter, to guess additional axioms, that avoid these infinitary behaviors. Such a guess is *speculative*, because it may cause an unsoundness, if it turns a satisfiable set into an unsatisfiable one. Thus, it must be *reversible*. DPLL($\Gamma + \mathcal{T}$) features a *SpeculativeIntro* rule, that adds an arbitrary clause C, written $\lceil C \rceil \triangleright C$, to F, and $\lceil C \rceil$ to M, where $\lceil C \rceil$ is a new propositional variable used to record the fact that the system is *guessing* C. If the guess turns out to be inconsistent, $\lceil C \rceil$, hence $\lceil C \rceil \triangleright C$, will be retracted by backjumping. Note that $\lceil C \rceil$ may end up in the hypotheses of clauses, hence $\neg\lceil C \rceil$ may appear in an asserting clause, recording a situation where $\lceil C \rceil$ is inconsistent. An unnatural failure due to a speculation, is treated in the same way as a natural failure due to the problem. Clearly, also the number of *SpeculativeIntro* steps is potentially unbounded, and therefore it is controlled by iterative deepening. DPLL($\Gamma + \mathcal{T}$) is said to be *stuck*, if it halts because the only possible inferences are Γ-inferences or speculative inferences that would violate their bounds.

In order to get a decision procedure, one needs to show that for some sequence of speculative axioms, there exists bounds of iterative deepening, on Γ-inferences and speculations, such that DPLL($\Gamma + \mathcal{T}$) is guaranteed to terminate in a state with the empty clause, whenever S is unsatisfiable, and in a non-conflict non-stuck state, whenever S is satisfiable. Continuing with the monotonicity example, let $\mathcal{R}$ be $\{\neg(x \sqsubseteq y) \vee \neg(y \sqsubseteq z) \vee x \sqsubseteq z,\ \neg(x \sqsubseteq y) \vee f(x) \sqsubseteq f(y)\}$, and P be $\{a \sqsubseteq b,\ a \sqsubseteq f(c),\ \neg(a \sqsubseteq c)\}$. If *SpeculativeIntro* adds $\lceil f(x) \simeq x \rceil \triangleright f(x) \simeq x$, monotonicity and $a \sqsubseteq f(c)$ are disabled by simplification, and $\lceil f(x) \simeq x \rceil \triangleright a \sqsubseteq c$ is added to F. Resolution generates the conflict clause $\lceil f(x) \simeq x \rceil \triangleright \Box$, so that $\neg\lceil f(x) \simeq x \rceil$ is added to M, preventing DPLL($\Gamma + \mathcal{T}$) from guessing $f(x) \simeq x$ again. Next, if *SpeculativeIntro* adds $\lceil f(f(x)) \simeq x \rceil \triangleright f(f(x)) \simeq x$, monotonicity and $a \sqsubseteq b$ produce only $f(a) \sqsubseteq f(b)$, while monotonicity and $a \sqsubseteq f(c)$ produce only $f(a) \sqsubseteq f(f(c))$, which is disabled and replaced by $\lceil f(f(x)) = x \rceil \triangleright f(a) \sqsubseteq c$. Then, DPLL($\Gamma + \mathcal{T}$) reaches a saturated state and detects satisfiability. In [26], this mechanism is shown to yield decision procedures for theories that are *essentially finite* – a generalization of the *finite model property* – and their combinations. This class includes axiomatizations of type systems, with either *single* or *multiple inheritance*, used in tools such as ESC/Java [51] and Spec# [13].

5. Discussion

A general objective in program checking is to increase

- *expressivity* of the logic, hence qualitative coverage of programs,
- *scalability* of performances, hence quantitative coverage of programs,
- *precision* of the results, hence reliability of the analysis, and
- *automation*, to reduce cost.

Different methods have different advantages and disadvantages with respect to these goals. For instance, *scalability* is regarded as a main challenge for software model checking, while static analysis may scale better, but at the expense of *precision*, which causes false alarms. Thus, a current trend is to *integrate* approaches, in order to leverage their strengths. While the cooperation of model checking and theorem proving has received significant attention, that of ab-

stract interpretation and theorem proving seems to have been less explored. A view of their interrelation was suggested in [112]. A question is whether and how the theorem prover could help to refine the abstraction itself, in addition to answer validity queries generated by a given abstraction. Other examples of integration include model checking and abstract interpretation, static analysis and directed testing, or static analysis and dynamic analysis (e.g., [98]). Integration poses both theoretical and engineering challenges, and it is problem-driven: it requires to choose which approaches to integrate for which classes of properties, such as safety properties, information flow properties, temporal properties. In addition to the integration of verification technologies, another evergreen quest is the cooperation of verification (i.e., checking that a program satisfies a property) and synthesis (i.e., generating a program that satisfies a property).

It is plausible that a cooperation of verification and synthesis happens for hardware sooner than for software. However, the constant evolution of the border between software and hardware is bound to challenge long-held classifications of problems and methods and favor fruitful hybridizations. Hardware design is more automated, employs fewer and more standardized languages; and hardware verification may employ simulation. The traditional conceptual difference is that a hardware circuit is modelled by a finite state machine, whereas a software program is modelled by an infinite state machine. Nonetheless, the transfer of functionalities from software to hardware, the use of software to design and produce hardware, and the growth of new contexts for computing (e.g., embedded systems, biological systems), means that approaches originally conceived for hardware may impact software and vice versa.

Integration is the keyword also in automated theorem proving. After several years where SMT-solvers and general theorem provers grew independently, the integration of general theorem prover within SMT-solver in DPLL($\Gamma + \mathcal{T}$) is a case in point. Symmetrically, approaches such as [1, 70] embed a $\mathcal{T}$-solver for linear arithmetic into a general theorem prover. In turn, the integration of automated and interactive theorem proving is more and more a reality. Induction is fundamental to reason about programs. For instance, in deductive verification as described here, there is a form of induction at the meta-level, in going from validity of the verification conditions for each basic path to validity of the specifications for the whole program. In general, since the set of inductive theorems is not even semi-decidable, inductive theorem proving involves some degree of interaction. However, the distinction itself between automated and interactive theorem proving is progressively fading. Automated theorem provers are interactive, as they require the user to set options, or even decorate the input with heuristic information such as triggers. Symmetrically, interactive theorem provers and proof assistants are automated, partly on their own, partly because they embed decision procedures, or invoke automated theorem provers, to discharge proof obligations generated as sub-proofs of the interactive proof. One could say that automated theorem provers have a long interaction cycle, and interactive theorem provers have a short interaction cycle.

This trend is likely to continue and become more general as theorem proving *disappears* inside applications. A fundamental reason for this to happen is that logic is increasingly proving to be as well adapted for machines as it is user-unfriendly for most humans. In essence, logic is a low-level language, and artificial intelligence may be more successful in enabling machines to think about their own circuits and programs than about most other subjects.

References

[1] E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic SUP(LA). In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the Seventh Symposium on Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Artificial Intelligence*, pages 84–99. Springer-Verlag, 2009.

[2] A. Armando. Building SMT-based software model checkers: an experience report. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the Seventh Symposium on Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Artificial Intelligence*, pages 1–17. Springer-Verlag, 2009.

[3] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2): 140–164, 2003.

[4] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. In B. Cook, S. Stoller, and W. Visser, editors, *Proceedings of the Third Workshop on Software Model Checking (SoftMC), Conference on Automated Verification (CAV) 2005*, volume 144(3) of *Electronic Notes in Theoretical Computer Science*, pages 79–94. Elsevier, 2006.

[5] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 10(1):129–179, 2009.

[6] L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *Journal of the ACM*, 41(2):236–276, 1994.

[7] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[8] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume II: Rewriting Techniques, pages 1–30. Academic Press, 1989.

[9] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.

[10] T. Ball. The SMT "Big Bang": applications of Z3 in Microsoft. Talk at the Dagstuhl Seminar 09411 Interaction versus automation: the two faces of deduction, 2009.

[11] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In J. C. Mitchell, editor, *Proceedings of the Twenty-Ninth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–3. ACM Press, 2002.

[12] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In M. L. Soffa, editor, *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213. ACM Press, 2001.

[13] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec♯ programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of the Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[14] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In M. Hermann and A. Voronkov, editors, *Proceedings of the Thirteenth International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer-Verlag, 2006.

[15] C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker. In R. Alur and D. A. Peled, editors, *Proceedings of the Sixteenth Conference on Automated Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, 2004.

[16] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In K. G. Larsen and E. Brinksma, editors, *Proceedings of the Fourteenth Conference on Automated Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, 2002.

[17] C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor,

Proceedings of the Fourth Workshop on Frontiers of Combining Systems (FroCoS), volume 2309 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[18] N. Bjørner and L. de Moura. SMT solvers in program analysis and verification. Tutorial at the Fourth International Joint Conference on Automated Reasoning (IJCAR), 2008.

[19] M. P. Bonacina. *Distributed Automated Deduction*. PhD thesis, Dept. of Computer Science, State University of New York at Stony Brook, 1992.

[20] M. P. Bonacina and N. Dershowitz. Abstract canonical inference. *ACM Transactions on Computational Logic*, 8(1):180–208, 2007.

[21] M. P. Bonacina and M. Echenim. Rewrite-based satisfiability procedures for recursive data structures. In B. Cook and R. Sebastiani, editors, *Proceedings of the Fourth Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR), Federated Logic Conference 2006*, volume 174(8) of *Electronic Notes in Theoretical Computer Science*, pages 55–70. Elsevier, 2007.

[22] M. P. Bonacina and M. Echenim. On variable-inactivity and polynomial T-satisfiability procedures. *Journal of Logic and Computation*, 18(1):77–96, 2008.

[23] M. P. Bonacina and M. Echenim. Theory decision by decomposition. *Journal of Symbolic Computation*, 45(2):229–260, 2010.

[24] M. P. Bonacina and J. Hsiang. Towards a foundation of completion procedures as semidecision procedures. *Theoretical Computer Science*, 146:199–242, 1995.

[25] M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and undecidability results for Nelson-Oppen and rewrite-based decision procedures. In U. Furbach and N. Shankar, editors, *Proceedings of the Third International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 513–527. Springer-Verlag, 2006.

[26] M. P. Bonacina, C. A. Lynch, and L. de Moura. On deciding satisfiability by DPLL($\Gamma+\mathcal{T}$) and unsound theorem proving. In R. Schmidt, editor, *Proceedings of the Twenty-Second Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 35–50. Springer-Verlag, 2009. Full version: `http://profs.sci.univr.it/~bonacina/dpllSPsi.html`.

[27] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning*, 35(1–3):265–293, 2005.

[28] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient theory combination via Boolean search. *Information and Computation*, 204(10):1493–1525, 2006.

[29] A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer-Verlag, 2007.

[30] R. Bruttomesso. *RTL Verification: From SAT to SMT(BV)*. PhD thesis, Università degli Studi di Trento, 2008.

[31] R. Chada and D. A. Plaisted. On the mechanical derivation of loop invariants. *Journal of Symbolic Computation*, 15(5–6):705–744, 1993.

[32] P. Chew. An improved algorithm for computing with equations. In *Proceedings of the Twenty-First Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 108–117. IEEE Computer Society Press, 1980.

[33] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the Twelfth Conference on Automated Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[34] P. Cousot. Abstract interpretation: achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness Int. Conf.* Scuola Superiore G. Reiss Romoli, L'Aquila, Italy, 2000. `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`.

[35] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Proceedings of the Fourth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.

[36] D. Cyrluk, O. Möller, and H. Rueß. An efficient decision procedure for a theory of fixed-sized bitvectors. In O. Grumberg, editor, *Proceedings of the Ninth Conference on Automated Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1997.

[37] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[38] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[39] L. de Moura and N. Bjørner. Efficient E-matching for SMT-solvers. In F. Pfenning, editor, *Proceedings of the Twenty-First Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 183–198. Springer-Verlag, 2007.

[40] L. de Moura and N. Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

[41] L. de Moura and N. Bjørner. Engineering DPLL(T) + saturation. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the Fourth International Joint Conference on Automated Reasoning (IJCAR)*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 475–490. Springer-Verlag, 2008.

[42] L. de Moura and N. Bjørner. Model-based theory combination. In S. Krstić and A. Oliveras, editors, *Proceedings of the Fifth Workshop on Satisfiability Modulo Theories (SMT), Conference on Automated Verification (CAV) 2007*, volume 198(2) of *Electronic Notes in Theoretical Computer Science*, pages 37–49. Elsevier, 2008.

[43] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of the Fourteenth Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.

[44] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In A. Biere and C. Pixley, editors, *Proceedings of the Ninth Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 45–52. IEEE Computer Society Press, 2009.

[45] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *Proceedings of the Eighteenth Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 438–455. Springer-Verlag, 2002.

[46] L. de Moura, S. Owre, H. Rueß, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In D. Basin and M. Rusinowitch, editors, *Proceedings of the Second International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 218–222. Springer-Verlag, 2004.

[47] N. Dershowitz and C. Kirchner. Abstract canonical presentations. *Theoretical Computer Science*, 357:53–69, 2006.

[48] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[49] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[50] B. Dutertre and L. de Moura. A fast linear arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Proceedings of the Eighteenth Conference on Automated Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 2006.

[51] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In L. J. Hendren, editor, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM Press, 2002.

[52] R. W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[53] P. Fontaine. Combinations of theories for decidable fragments of first-order logic. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the Seventh Symposium on Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Artificial Intelligence*, pages 263–278. Springer-Verlag, 2009.

[54] J. Gallier, P. Narendran, D. A. Plaisted, S. Raatz, and W. Snyder. Finding canonical rewriting systems equivalent to a finite set of ground equations in polynomial time. *Journal of the ACM*, 40(1): 1–16, 1993.

[55] H. Ganzinger, H. Rueß, and N. Shankar. Modularity and refinement in inference systems. Technical Report CSL-SRI-04-02, SRI International, 2004.

[56] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In F. Pfenning, editor, *Proceedings of the Twenty-First Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 167–182. Springer-Verlag, 2007.

[57] S. Ghilardi. Model-theoretic methods in combined constraints satisfiability. *Journal of Automated Reasoning*, 33:221–249, 2004.

[58] S. Graf and H. Säidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proceedings of the Ninth Conference on Automated Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[59] A. Gupta. Software verification: rôles and challenges for automatic decision procedures. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the Fourth International Joint Conference on Automated Reasoning (IJCAR)*, volume 5195 of *Lecture Notes in Artificial Intelligence*, page 1. Springer-Verlag, 2008.

[60] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In J. C. Mitchell, editor, *Proceedings of the Twenty-Ninth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM Press, 2002.

[61] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In X. Leroy, editor, *Proceedings of the Thirty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 232–244. ACM Press, 2004.

[62] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[63] C. A. R. Hoare. The verifying compiler: a grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[64] J. Hsiang and M. Rusinowitch. On word problems in equational theories. In T. Ottman, editor, *Proceedings of the Fourteenth International Colloquium on Automata Languages and Programming (ICALP)*, volume 267 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 1987.

[65] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem proving strategies: the transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, 1991.

[66] H. Jain. *Verification using satisfiability checking, predicate abstraction and Craig interpolation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2008.

[67] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In P. Devambu, editor, *Proceedings of the Fourteenth ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM Press, 2006.

[68] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[69] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Proceedings of the Conference on Computational Problems in Abstract Algebras*, pages 263–298. Pergamon Press, 1970.

[70] K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In J. Duparc and T. A. Henzinger, editors, *Proceedings of the Sixteenth EACSL Annual Conference on Computer Science Logic (CSL)*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer-Verlag, 2007.

[71] E. Kounalis and M. Rusinowitch. On word problems in Horn theories. *Journal of Symbolic Computation*, 11(1–2):113–128, 1991.

[72] L. Kovacs and A. Voronkov. Interpolation and symbol elimination. In R. Schmidt, editor, *Proceedings of the Twenty-Second Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 199–213. Springer-Verlag, 2009.

[73] L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In M. Chechik and M. Wirsing, editors, *Proceedings of the Twelfth Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *Lecture Notes in Computer Science*, pages 470–485. Springer-Verlag, 2009.

[74] S. Krstić and A. Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In F. Wolter, editor, *Proceedings of the Sixth Symposium on Frontiers of Combining Systems (FroCoS)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 1–27. Springer-Verlag, 2007.

[75] S. Lahiri and S. Qaader. Back to the future: Revisiting precise program verification using SMT solvers. In G. C. Necula and P. Wadler, editors, *Proceedings of the Thirty-Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 171–182. ACM Press, 2008.

[76] S. Lahiri and S. Qaader. Verification. Tutorial at the Twenty-Second Conference on Automated Deduction, 2009.

[77] D. S. Lankford. Canonical inference. Memo ATP-32, Automatic Theorem Proving Project, University of Texas at Austin, 1975.

[78] J.-L. Lassez and M. J. Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning*, 9(3):373–379, 1992.

[79] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[80] K. R. M. Leino and W. Schulte. A verifying compiler for a multithreaded object-oriented language. To appear in the Marktoberdorf Summer School 2006 lecture notes, 2006.

[81] J. McCarthy. Towards a mathematical science of computation. In *International Federation for Information Processing*, pages 21–28. 1962.

[82] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. 1963.

[83] W. W. McCune. Otter 3.3 reference manual. Technical Report ANL/MCS-TM-263, MCS Division, Argonne National Laboratory, 2003.

[84] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[85] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of the Fourteenth Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 413–427. Springer-Verlag, 2008.

[86] B. Meyer. The grand challenge of trusted components. In *Proceedings of the Twenty-Fifth International Conference on Software Engineering*. IEEE Computer Society Press, 2003.

[87] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In D. Blaauw and L. Lavagno, editors, *Proceedings of the Thirty-Ninth Design Automation Conference (DAC)*, pages 530–535, 2001.

[88] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[89] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[90] E. Nicolini, C. Ringeissen, and M. Rusinowitch. Data structures with arithmetic constraints: a non-disjoint combination. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the Seventh Symposium on Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lec-*

ture Notes in Artificial Intelligence, pages 319–334. Springer-Verlag, 2009.

[91] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

[92] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 371–443. Elsevier, 2001.

[93] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[94] D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the ACM*, 27(3), 1980.

[95] D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.

[96] D. A. Plaisted. Abstraction using generalization functions. In J. H. Siekmann, editor, *Proceedings of the Eighth Conference on Automated Deduction (CADE)*, volume 230 of *Lecture Notes in Computer Science*, pages 365–376. Springer-Verlag, 1986.

[97] D. A. Plaisted and Y. Zhu. *The Efficiency of Theorem Proving Strategies*. Friedr. Vieweg & Sohns, 1997. Second edition: GWV-Vieweg, 1999.

[98] S. K. Rajamani. Static and dynamic analysis: better together. In Z. Shao, editor, *Proceedings of the Seventh Asian Symposium on Programming Languages and Systems (APLAS)*, volume 4807 of *Lecture Notes in Computer Science*, page 302. Springer-Verlag, 2007.

[99] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *Journal of AI Communications*, 15(2/3):91–110, 2002.

[100] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In D. Michie and R. Meltzer, editors, *Machine Intelligence*, volume IV, pages 135–150. Edinburgh University Press, 1969.

[101] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[102] H. Rueß and N. Shankar. Deconstructing Shostak. In J. Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2001.

[103] M. Rusinowitch. Theorem-proving with resolution and superposition. *Journal of Symbolic Computation*, 11:21–50, 1991.

[104] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[105] S. Schulz. E – A brainiac theorem prover. *Journal of AI Communications*, 15(2–3):111–126, 2002.

[106] R. Sebastiani. Lazy satisfiability modulo theories. *Journal of Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[107] R. E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, 1978.

[108] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[109] W. Snyder. A fast algorithm for generating reduced ground rewriting systems from a set of ground equations. *Journal of Symbolic Computation*, 1992.

[110] A. Stump, C. W. Barrett, D. L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In J. Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2001.

[111] C. Tinelli and M. Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In F. Baader and K. Schulz, editors, *Proceedings of the First Workshop on Frontiers of Combining Systems (FroCoS)*, volume 3 of *Applied Logic Series*. Kluwer, 1996.

[112] A. Tiwari and S. Gulwani. Logical interpretation: static program analysis using theorem proving. In F. Pfenning, editor, *Proceedings of the Twenty-First Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 147–166. Springer-Verlag, 2007.

[113] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in constructive mathematics and mathematical logic*, volume 2, pages 115–125. Consultants Bureau, 1970. Reprinted in J. Siekmann and G. Wrightson (eds.), *Automation of Reasoning*, Vol. 2, 466–483, Springer, 1983.

[114] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. Schmidt, editor, *Proceedings of the Twenty-Second Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145. Springer-Verlag, 2009.

[115] T. Wies, R. Piskac, and V. Kuncak. Combining theories with shared set operations. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the Seventh Symposium on Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Artificial Intelligence*, pages 366–382. Springer-Verlag, 2009.

[116] L. Wos, G. Robinson, D. Carson, and L. Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14(4):698–709, 1967.

[117] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In R. Nieuwenhuis, editor, *Proceedings of the Twentieth Conference on Automated Deduction (CADE)*, volume 3632 of *Lecture Notes in Artificial Intelligence*, pages 353–368. Springer-Verlag, 2005.

[118] H. Zhang and M. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1–2):277–296, 2000.

[119] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In A. Voronkov, editor, *Proceedings of the Eighteenth Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 295–313. Springer-Verlag, 2002.