Automated Deduction for Verification

Natarajan Shankar
SRI International

Automated deduction uses computation to perform symbolic logical reasoning. It has been a core technology for program verification from the very beginning. Satisfiability solvers for propositional and first-order logic significantly automate the task of deductive program verification. We introduce some of the basic deduction techniques used in software and hardware verification and outline the theoretical and engineering issues in building deductive verification tools. Beyond verification, deduction techniques can also be used to support a variety of applications including planning, program optimization, and program synthesis.

Categories and Subject Descriptors: F41 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; I23 [**Artificial Intelligence**]: Deduction and Theorem Proving

General Terms: Theory, Theorem Proving, Formal Verification

Additional Key Words and Phrases:

1. INTRODUCTION

The feasibility of large-scale formal software verification rests squarely on the development of robust, sophisticated, and scalable verification tools. Recent advances in verification technology on a number of fronts have made it possible to contemplate a major push toward large-scale software verification [Hoare 2003; Hoare and Misra 2008]. These advances have already yielded practical tools for solving hard verification problems. Many of these tools are already in industrial use. Deductive techniques are used both for finding bugs and for stating and proving correctness properties. They can also be used to construct and check models and proofs, and to synthesize functions and formulas in a wide range of logical formalisms. We survey deductive approaches to verification based on satisfiability solving, automated proof search, and interactive proof checking. We examine some of the recent progress and outline a few of the most promising avenues for dramatic improvements in the technologies of verification.

Automated theorem provers and interactive proof checkers have been associated with verification from the beginning [King 1969; King and Floyd 1970]. Jones [1992] covers the history of verification research. McCarthy [1963], who initiated the formal study of program verification, was also involved in the construction of one of the early proof check-

Funded by NSF CISE Grant Nos. 0646174 and 0627284, NSF SGER grant CNS-0823086, and NASA Cooperative Agreement NNX08AY53A. The views and opinions expressed in this document are not necessarily those of the National Science Foundation, the National Aeronautics and Space Administration, or any other agency of the United States Government.

Address: SRI International Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025. Email: shankar@csl.sri.com. Phone: (650)859-5272.

ers. The early nexus between verification and deduction is beautifully surveyed by Elspas, Levitt, Waldinger, and Waksman [1972]. Hantler and King [1976] give a brief overview of the early results in program verification.

In the 1970s, several research groups began working on the use of theorem provers in verification. These included the Gypsy project at the University of Texas at Austin [Smith et al. 1988]; the Affirm project [Gerhart et al. 1980] at USC-ISI; the FDM project at System Development Corporation; the Jovial Verifier [Elspas et al. 1979] and HDM [Robinson et al. 1979], and STP [Shostak et al. 1982] at SRI International; the Boyer–Moore prover [Boyer and Moore 1979] (initiated at the University of Edinburgh and later continued at Xerox PARC, SRI International, and the University of Texas at Austin); the LCF (Logic for Computable Functions) project [Gordon et al. 1979] (initiated at Stanford University and continued at the University of Edinburgh and the University of Cambridge); and the FOL [Weyhrauch 1980] and Stanford Pascal Verifier [Luckham et al. 1979] projects at Stanford University.

The basic problem in deduction is whether a statement is *valid*. If a statement has a *proof* in a *sound* proof system, then it is valid, and if the proof system is *complete*, then every valid statement does have a proof. If a statement is not valid, then it has a *counterexample*. In other words, its negation is *satisfiable*. The validity problem is *decidable* if there is a program that can, in principle, determine if a given statement is valid. Some logics are *semi-decidable* so that there is a program that does terminate with the right result when the given statement is valid, but it might not terminate on invalid statements.

The first part of the survey (Section 2) is a brief introduction to the logical formalisms that are used in automated deduction including propositional logic, equational logic, first-order logic, set theory, and higher-order logic. The second part (Section 3) covers *satisfiability* procedures for propositional logic and fragments of first-order logic including theories such as linear arithmetic, arrays, and bit vectors. Satisfiability procedures can be used as decision procedures to decide whether a given formula is valid. These are the workhorses of deductive verification. We then move on to *proof search* procedures (Section 4) which typically work in domains where the validity problem is not decidable. Most such procedures are *uniform*: the theory axioms are given as explicit inputs without assuming any special knowledge about the theory-specific symbols. In contrast, satisfiability procedures typically build in theory-specific techniques. Most verification problems tend to be theory intensive so that the applicability of uniform methods has been limited to problems that make heavy use of quantification. The third part of our survey (Section 5) describes interactive proof engines that can be used to develop a large body of mathematics through a sequence of definitions and lemmas. Proofs are developed interactively with varying degrees of automated support. Both satisfiability and proof search procedures can be usefully employed as automated tools in interactive proof construction. We conclude with some observations about the promising research directions for automated deduction.

Automated deduction tools can be used in a variety of ways in formal verification in applications ranging from modeling requirements and capturing program semantics to generating test cases and validating compiler optimizations. Automated deduction can be used to

(1) Capture and analyze software specifications and properties [Jones 1990; Spivey 1993].
(2) Formalize the mathematical background and domain knowledge used in reasoning about program correctness.

(3) Define the semantics of programming languages and logics [Gordon 1989].

(4) Discharge verification, termination and type correctness conditions [Naur 1966; Floyd 1967; King 1969; Owre et al. 1995; Detlefs et al. 1998; Gulwani and Tiwari 2006].

(5) Synthesize programs from the specification [Manna and Waldinger ; Darlington 1981] or to refine them in stages to yield executable software [Abrial 1996; Smith 1990].

(6) Implement the operations needed to compute symbolic fixed points for finite and infinite-state systems [Cousot and Cousot 1977; Burch et al. 1992; Abdulla et al. 1996; Bultan et al. 1997; Delzanno and Podelski 2001; Jhala and Majumdar 2009].

(7) Construct and refine abstractions of systems guided by counterexamples [Saïdi and Graf 1997; Kesten and Pnueli 1998; Ball et al. 2001; Clarke et al. 2003; Wies et al. 2006].

(8) Generate test cases by solving constraints arising from the symbolic execution of a program [Boyer et al. 1975; King 1976; Clarke 1976; Hamon et al. 2004; Godefroid et al. 2005; Hierons et al. 2009].

These applications share many of the same basic deductive techniques. Automated deduction is a vast and growing field and there are hundreds of systems that support logic-based automated verification. Our survey covers a cross-section of the deductive techniques and tools that are used in verification, with a particular emphasis on satisfiability solvers. The exposition here is directed at non-experts who are interested in a deeper understanding of the techniques of automated deduction. We mainly focus on those techniques that are relevant for formal verification. A recent book by Harrison [2009] contains a more comprehensive modern treatment of automated deduction.

2. MATHEMATICAL LOGIC

A strong facility with logic is an essential skill for a computer scientist. We review formal logic from the point of view of automated and semi-automated verification. We briefly introduce propositional logic, equational logic, first-order logic, and higher-order logic. A reader familiar with these topics can safely skip over this introduction. Good introductions to logic are available from several sources including Barwise's article *An Introduction to First-Order Logic* [Barwise 1978b], and books by Kleene [1952; 1967]; Shoenfield [1967]; Boolos and Jeffrey [1989]; Enderton [1972]; Mendelson [1964]; van Dalen [1983]; Fitting [1990]; Girard, Lafont, and Taylor [1989]; and Ebbinghaus, Flum, and Thomas [1984]. The topic of logic in computer science is well covered in the book by Huth and Ryan [2000]. There are several handbooks including those on Mathematical Logic [Barwise 1978a], Philosophical Logic [Gabbay and Guenthner 1983; 1984; 1985], Logic in Computer Science [Abramsky et al. 1992a; 1992b], Theoretical Computer Science [van Leeuwen 1990], Automated Reasoning [Robinson and Voronkov 2001], Tableau Methods [D'Agostino et al. 1999], and a recent one on Satisfiability [Biere et al. 2009].

David Gries and Fred Schneider [1993] have observed that *logic is the glue that binds together methods of reasoning, in all domains*. Many different domains can be related through their interpretation within logic. Inference procedures for logic can be applied to these embedded formalisms. Logic has been "unreasonably effective" in computer science [Halpern et al. 2001; Feferman 2006] with deep connections to computability, complexity, database theory, hardware design, and programming language semantics, as well as the formal specification and verification of hardware and software.

Mathematical logic is basic to the operation of verification tools. Verification tools make formal claims about software. We need a language in which these claims are expressed. We also need a calculus in which these claims are justified and combined to yield new claims. Logic is the calculus of computing. Within the purview of logic, there is a wide range of formalisms for dealing with different aspects of software. First, there is *propositional logic*, where the expressions are built from propositional variables using the connectives for conjunction, disjunction, negation, implication, and equivalence. Various *modal* and *temporal* logics extend propositional logic to reason about modalities like time, necessity, knowledge, and belief over propositions. *First-order logic* extends propositional logic to predicates and terms built from variables and function symbols, and serves as a formal foundation for arithmetic and set theory. *Equational logic* is a fragment of first-order logic that provides the foundation for algebraic reasoning using equalities. *Higher-order logic* allows quantification over functions and predicates and is suitable for modeling computation at varying levels of abstraction and for formalizing much of classical mathematics.

A logic consists of a formal language, a formal semantics, and a formal proof system. The language captures the rules for forming statements and circumscribes the range of concepts that can be expressed. The *formal semantics* defines the intended interpretation of the symbols and expressions used in these statements. The formal semantics can be used to identify different expressions that have the same interpretation. It fixes the meaning of certain symbols, e.g., the logical connectives and allows the meaning of other symbols, e.g., variables and functions, to vary within certain bounds. The formal proof system is a framework of rules for deriving valid statements. While the textbook presentation of a proof system is usually minimalist, any practical system will employ quite sophisticated proof rules that can be justified in foundational terms. Many practical proof checking systems also allow new proof rules to be added when they can be explicitly justified in terms of existing ones.

Logic can be used in all kinds of interesting ways. It can highlight the limitations of a formal language by demonstrating that certain concepts are not definable in it. It can of course be used to prove theorems, and this is the use that will be most interesting here. Logic also has a dual use which is to generate concrete instances (models) of a given formula as in planning, constraint solving, and test case generation. At the metatheoretic level, relationships between logics can be used to map results from one logic to another, or to reduce problems from one logic to another [Tarski et al. 1971; Meseguer 1989].

2.1 Propositional Logic

Propositional logic plays an important role in digital hardware design as well as hardware and software verification. A *proposition* is a declarative statement such as "Mary has a book". In propositional logic, the propositions are atomic so that "Mary has no book" is a separate proposition that is unrelated to the first. Basic propositions are just represented by propositional atoms or variables. Propositional formulas are built from propositional atoms using the logical operators $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\Rightarrow$ (implication), and $\Leftrightarrow$ (equivalence). In the classical interpretation, propositional formulas are evaluated with respect to a Boolean truth assignment of $\top$ or $\bot$ to the atoms. A formula is satisfiable if there is some truth assignment under which the formula evaluates to $\top$. Thus $p \wedge (p \Rightarrow q)$ is satisfied by the truth assignment $\{p \mapsto \top, q \mapsto \top\}$. On the other hand, $p \wedge \neg p$ is not satisfiable. If a formula is unsatisfiable, then its negation is valid, i.e., evaluates to $\top$ under any assignment of truth values to the atoms. Since a formula has finitely many

x	y	$\underline{\neg}x$	$x\underline{\vee}y$
$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$		$\top$
$\top$	$\bot$	$\bot$	$\top$
$\top$	$\top$		$\top$

Fig. 1. Truth table semantics for $\neg$ and $\vee$

distinct propositional atoms, each of which has two possible truth values, the satisfiability of a formula with n atoms can obviously be decided by evaluating the formula on the 2^n possible truth assignments. Later, we will examine more refined methods for finding satisfying truth assignments or showing that the formula is unsatisfiable. However, since the satisfiability problem is NP-complete, there is no known sub-exponential algorithm for it.

Language and Semantics. For our purpose, a propositional formula ϕ is either an atom from a set $\mathcal{A}$ of atoms, a negation $\neg\phi_1$, or a disjunction $\phi_1 \vee \phi_2$. Conjunction, implication, and equivalence can be easily defined from negation and disjunction. A *structure* or a *truth assignment* M maps atoms in $\mathcal{A}$ to truth values from the set $\{\top, \bot\}$. The truth table semantics for the connectives is given in Figure 1 by defining interpretations of $\underline{\neg}$ and $\underline{\vee}$.

The interpretation of a formula ϕ with respect to a structure M is given by $M[\![\phi]\!]$ as defined below.

$$\begin{aligned} M[\![p]\!] &= M(p), \text{ for } p \in \mathcal{A} \\ M[\![\neg\phi]\!] &= \underline{\neg} M[\![\phi]\!] \\ M[\![\phi_1 \vee \phi_2]\!] &= M[\![\phi_1]\!] \underline{\vee} M[\![\phi_2]\!] \end{aligned}$$

A structure M is a *model* for a formula ϕ, i.e., $M \models \phi$, if $M[\![\phi]\!] = \top$. A formula ϕ is *satisfiable* if for some structure M, $M \models \phi$. A formula ϕ is *valid* if for all structures M, $M \models \phi$. For example, the formula $p \wedge \neg q$ is satisfied in the model $\{p \mapsto \top, q \mapsto \bot\}$. The formula $p \vee \neg p$ is valid, and its negation $\neg(p \vee \neg p)$ is unsatisfiable.

Normal Forms. Formulas can be transformed into equivalent forms that are more convenient for syntactic manipulation. We first introduce conjunction into the language, a formula in classical propositional logic. The negation normal form (NNF) applies rules like $\neg(p \wedge q) = \neg p \vee \neg q$, $\neg(p \vee q) = \neg p \wedge \neg q$, and $\neg\neg p = p$ to ensure that only atoms can appear negated. An atom or its negation is termed a *literal*. A *clause* is a disjunction of literals. Any formula can be converted to conjunctive normal form (CNF) where it appears as a conjunction of clauses. Dually, a formula can also be expressed in disjunctive normal form (DNF) where it appears as a disjunction of *cubes*, where a cube is a conjunction of literals. For example, the formula $\neg(p \vee q) \vee \neg(\neg p \vee \neg q)$ can be converted into the NNF $(\neg p \wedge \neg q) \vee (p \wedge q)$. The latter formula is already in DNF. It can be converted into CNF as $(\neg q \vee p) \wedge (\neg p \vee q)$, which happens to be the $(\neg, \vee)$-formula representing $p \Leftrightarrow q$. The conversion of a propositional formula to an equisatisfiable CNF can be done in linear time by introducing new propositional atoms to represent subformulas. Practical algorithms for CNF conversion try to minimize the number of clauses generated by identifying equivalent subformulas [Tseitin 1968; Jackson and Sheridan 2004; Manolios and Vroon 2007].

Intuitionistic Logic. In contrast to the classical interpretation of the logical connectives described above, intuitionistic logic [Troelstra and van Dalen 1988] disallows the classically valid rules of the excluded middle $p \vee \neg p$ and double negation elimination $\neg\neg p \Rightarrow p$. Whereas classical logic is about proving that a formula is valid in all interpretations, intuitionistic logic is about supporting the conclusion with actual evidence. Thus, $p \vee \neg p$ is classically valid, but evidence for a disjunction must be either evidence for p or for $\neg p$, and we do not have such evidence. Similarly, evidence for $\neg p$ shows that any evidence for p can be used to construct a contradiction. Then evidence for $\neg\neg p$ demonstrates the absence of evidence for $\neg p$, which is not taken as evidence for p. The excluded middle rule allows non-constructive proofs of existence as is illustrated by the following demonstration that there exist irrational numbers x and y such that x^y is rational. Either $\sqrt{2}^{\sqrt{2}}$ is rational, in which case x and y can both be taken as $\sqrt{2}$, or we pick x to be $\sqrt{2}^{\sqrt{2}}$ and y to be $\sqrt{2}$ so that x^y is just $\sqrt{2}^2$ which simplifies to 2. We have demonstrated the existence of x and y without providing a construction since we do not have an effective way of determining whether $\sqrt{2}^{\sqrt{2}}$ is rational. We survey some interactive proof checkers for intuitionistic proofs in Section 5.3.2.

Proof systems for propositional logic. In *Hilbert-style* proof systems, each inference rule has zero or more *premise* formulas and one *conclusion* formula, and a proof is a tree of inference rule applications. The rule of *modus ponens* which derives $\vdash \psi$ from the premises $\vdash \phi$ and $\vdash \phi \Rightarrow \psi$ is a typical Hilbert-style rule. In the *natural deduction* style due to Gentzen, proof rules involve conditional judgments that assert the derivability of a consequent formula from some assumption formulae. The inference rules of natural deduction indicate how compound consequent formulas such as conjunctions, disjunctions, and implications are introduced or eliminated in a proof. Examples of natural deduction rules are shown in Figure 13 and the exact form of these rules is explained in Section 5.3.2. Gentzen's *sequent calculus* shown in Figure 2 has a Hilbert-style inference form but the premises and conclusions are *sequents*, which are of the form $\Gamma \vdash \Delta$, where Γ is a finite set of *antecedent* formulas and Δ is a finite set of *consequent* formulas. A sequent $\Gamma \vdash \Delta$ asserts that the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ. Hence, if a sequent is not valid, then there is an interpretation under which all the formulas in Γ evaluate to $\top$ and all the formulas in Δ evaluate to $\bot$. Figure 2 displays the rules for $\Rightarrow$ and $\wedge$. To each connective, there is an introduction rule for antecedent formulas (left rule) and another one for consequent formulas (right rule). The rules for $\Rightarrow$ and $\wedge$ could easily be derived from their definitions in terms of $\neg$ and $\vee$, but in the sequent calculus, the connectives are defined by their proof rules.

For example, the validity of Peirce's law $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$ in classical propositional logic can be demonstrated semantically by the truth-table method, or it can be proved in the sequent calculus.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p, q}\,Ax}{\vdash p, p \Rightarrow q}\,{\vdash\Rightarrow} \qquad \dfrac{}{p \vdash p}\,Ax}{(p \Rightarrow q) \Rightarrow p \vdash p}\,{\Rightarrow\vdash}}{\vdash ((p \Rightarrow q) \Rightarrow p) \Rightarrow p}\,{\vdash\Rightarrow}$$

	Left	Right
Ax	$\frac{}{\Gamma, A \vdash A, \Delta}$	
$\neg$	$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$	$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$
$\vee$	$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$	$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$
$\wedge$	$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$	$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$
$\Rightarrow$	$\frac{\Gamma, B \vdash \Delta \quad \Gamma \vdash A, \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}$	$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}$
Cut	$\frac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}$	

Fig. 2. A Sequent Calculus for Propositional Logic

The cut rule of the sequent calculus is *admissible* in terms of the remaining inference rules, i.e., if the premises are provable, then so is the conclusion of the cut rule. Note that the cut rule is not *derivable*: the conclusion cannot be proved from the premises without the use of the cut rule. Every derivable rule is also admissible. The equivalence rule

$$\frac{p \Leftrightarrow q}{\phi \Leftrightarrow \phi\{p \mapsto q\}} ,$$

where $\phi\{p \mapsto q\}$ is the result of substituting the proposition q for p in ϕ is an example of a derived rule.

An intuitionistic sequent calculus is obtained from the classical one by restricting the consequents in any sequent in a proof to at most one formula. It can be checked for example that Peirce's formula is not provable with this restriction since the proof requires a sequent of the form $\vdash p, p \Rightarrow q$.

Soundness and Completeness. For *soundness*, every provable statement must be valid. For the sequent calculus, this can be shown by induction on proofs since each axiom is valid and each proof rule asserts a valid conclusion when given valid premises. Note that a sequent $\Gamma \vdash \Delta$ is valid if for any interpretation M, either $M \not\models \gamma$ for some $\gamma \in \Gamma$ or $M \models \delta$ for some $\delta \in \Delta$.

A proof calculus is *complete* if every valid statement is provable. In particular, if formula ϕ does not have a proof, then there is an M such that $M \models \neg\phi$. There are several ways to prove completeness. One way is to start with a set H which is initially set to $\{\neg\phi\}$ and an enumeration of the formulas ψ_i, $i \geq 0$. We say that a $(\neg, \vee)$-formula ψ is *consistent* with H if $\not\vdash (\bigvee_{\theta \in H} \neg\theta) \vee \neg\psi$ in the proof calculus. For each, ψ_i, if ψ_i is consistent with H, we add ψ_i to H, and otherwise we add $\neg\psi_i$. We can then check with respect to the proof system that

(1) If $\psi \in H$ then $\neg\psi \notin H$
(2) If $\neg\psi \in H$ then $\psi \notin H$
(3) If $\psi \vee \psi' \in H$, then either $\psi \in H$ or $\psi' \in H$.

Note that for each atom p, either p or $\neg p$ is in H. We construct the model M_H so that

$M_H(p) = \top \iff p \in H$. It is easy to check that for each formula $\psi \in H$, $M_H \models \psi$. In particular, we have $M_H \models \neg\phi$.

Modal Logics. Propositional logic captures reasoning over truth assignments to the propositions. In particular, a statement is valid if it holds for all possible truth assignments. Modal logics admit modal operators for possibility, belief, and time that are indexed by truth assignments (worlds). A Kripke model consists of a set of worlds with an accessibility relation between worlds. The modality $\Box\phi$ when evaluated in a world w signifies that ϕ holds in all the worlds w' accessible from w. Dually, $\Diamond\phi$ holds in w if there is an accessible w' where ϕ holds. Most modal logics contain the inference rules of *modus ponens*, substitution, and necessitation

$$\frac{p}{\Box p}.$$

They also satisfy the distributivity axiom **K**: $\Box(p \Rightarrow q) \Rightarrow \Box p \Rightarrow \Box q$. With a reflexive accessibility relation, we get the logic **T** corresponding to the axiom $\Box p \Rightarrow p$. A reflexive and transitive accessibility relation yields **S4** with the added axiom $\Box p \Rightarrow \Box\Box p$. The modal logic **S5**, where the accessibility relation is an equivalence relation (i.e., reflexive, symmetric, and transitive), is obtained by adding the axiom $p \Rightarrow \Box\Diamond p$ to **S4**. Naturally, the valid formulas of **T** are a subset of those of **S4**, which are in turn a subset of those of **S5**.

Several modal logics are widely used in formal verification including those for linear time LTL, branching time CTL, the logic CTL* which combines branching and linear time logics, interval temporal logics, real-time temporal logics, dynamic logic, epistemic logics, and deontic logics. Satisfiability procedures for modal logic can be constructed either directly or through a translation into another decidable logic such as the *guarded* fragment of first-order logic where the quantified variables must appear in an atomic guard formula, or to the weak second order logic of two successors (WS2S) [Ohlbach et al. 2001]. Modal and temporal logics are surveyed by Goldblatt [1992], Mints [1992], Emerson [1990], and Blackburn, de Rijke, and Venema [2002].

Applications. Propositional logic has innumerable applications since any problem of bounded over a bounded domain can be expressed in it and the resulting formula can be solved for satisfiability. Solvable variants of satisfiability include the problems of generating all satisfying solutions (AllSAT), *unsatisfiable cores* consisting of the smallest unsatisfiable subset of input clauses, and the largest subset of satisfiable clauses (MAXSAT), are also solvable.

Propositional logic can be used to model electrical circuits by representing the presence of a high voltage on a wire as a proposition. A half adder with inputs a and b and output s_{out} can be represented as $s_{out} = (a \oplus b)$, where $\oplus$ is the exclusive-or operator so that $a \oplus b$ is defined as $(a \Rightarrow \neg b) \wedge (\neg b \Rightarrow a)$. An n-bit adder that adds two n-bit bit-vectors $\vec{a}$ and $\vec{b}$ with a carry-in bit c_{in} to produce an n-bit sum $\vec{s}_{out}$ and a carry-out bit c_{out} can be defined using the half adder to produce the sum and carry in a bit-wise manner.

Constraints over finite domains can be expressed in propositional logic. For example, the pigeonhole principle asserts that is impossible to assign $n+1$ pigeons to n holes so that there is at most one pigeon per hole. For this, we need $n(n+1)$ atoms p_{ij} for $0 \leq i < n$ and $0 \leq j \leq n$ expressing the proposition that the i'th hole holds the j'th pigeon. We can assert that each pigeon is assigned a hole as $\bigwedge_{j=0}^{n} \bigvee_{i=0}^{n-1} p_{ij}$. The constraint that no hole contains more than one pigeon is expressed as $\bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{n} \bigwedge_{k<j} (\neg p_{ij} \vee \neg p_{ik})$.

Planning is another application of constraint solving in the Boolean domain [Kautz and Selman 1996]. Consider the problem of planning truck routes to transfer packages between cities given the initial location of each truck and its fuel gauge reading, the source and destination of each package, the routes connecting the cities with the fuel needs, and the locations of the gas stations. The goal is to find the shortest plan that gets the packages delivered where in each step of the plan, a truck can load a package, unload a package, fill gas, or drive between two adjacent cities.

Scheduling is similar to planning and can also be encoded by means of a propositional formula. For example, consider the problem of constructing a timetable for a sports league consisting of n teams that must each play $n/2$ home games and $n/2$ away games so that each team plays every other team at least once and never has more than two away games in a row.

Program behavior for programs where the program state can be encoded using a bounded number of bits can also be modeled using propositional logic [Kroening et al. 2003]. For example, a program which computes the absolute value of a 32-bit two's-complement word can be verified purely in Boolean terms. That is, given a procedure abs on a 32-bit bit-vector $\vec{x}$, we can check that

$$\vec{y} = abs(\vec{x}) \Rightarrow (\vec{y} \geq 0 \wedge (\vec{y} = \vec{x} \vee \vec{y} = -\vec{x})).$$

More generally, a state of bounded size can be represented as a bit-vector of length n. The initial state of the program can be specified by a state predicate I. The (possibly nondeterministic) transition relation for the program can be encoded as a relation N on the state bit-vectors. In *bounded model checking* [Biere et al. 1999; D'Silva et al. 2008], we want to check that the property P is not violated within k steps by verifying that

$$I(\vec{x}_0) \wedge \bigwedge_{i=0}^{k-1} N(\vec{x}_i, \vec{x}_{i+1}) \wedge (\bigvee_{j=0}^{k} \neg P(\vec{x}_j))$$

is not satisfiable. If this formula turns out to be satisfiable, we can construct a sequence of states $\xi_0, \ldots, \xi_k$ from the Boolean assignment to each Boolean variable $\vec{x}_i[j]$, for $0 \leq i \leq k$ and $0 \leq j < n$. The Cook-Levin theorem [1971; 1973] showing that polynomial-bounded Turing machine computations can be reduced in polynomial time to propositional satisfiability can be seen as form of bounded model checking of Turing machines over $p(n)$ steps with a state consisting of an array of $2p(n)$ tape symbols, a control state, and a head position.

A state predicate P is an *invariant* for the transition system $\langle I, N\rangle$ if for any infinite state sequence ξ of the form $\langle \xi_0, \xi_1, \xi_2, \ldots\rangle$, the assertion $I(\xi_0) \wedge ((\forall i. N(\xi_i, \xi_{i+1})) \Rightarrow \forall i. P(\xi_i)$ is valid. We can check that a predicate P is *inductive* by verifying that $I(\vec{x}) \Rightarrow P(\vec{x})$ and $P(\vec{x}) \wedge N(\vec{x}, \vec{x}') \Rightarrow P(\vec{x}')$ are both valid. An inductive predicate is an invariant, but not all invariants are inductive. Invariant checking can be strengthened through the use of k-induction [Sheeran et al. 2000] to check that state predicate P is *k-inductive*. Here, the base case is exactly the bounded model checking step above, and the induction step is

$$\bigwedge_{i=0}^{k} N(\vec{x}_i, \vec{x}_{i+1}) \wedge (\bigwedge_{j=0}^{k} P(\vec{x}_j)) \Rightarrow P(\vec{x}_{k+1}).$$

A k-inductive predicate is also an invariant.

For symbolic model checking [Burch et al. 1992; McMillan 1993; Clarke et al. 1999], the set of reachable states is computed by representing each iteration as a propositional formula R_i so that $R_0 = I(\vec{x})$ and $R_{i+1} = image(N(\vec{x}, \vec{x}'))(R_i)$, where $image(\phi)(\psi) = (\exists \vec{x}''.(\phi \wedge \psi)\{\vec{x} \mapsto \vec{x}''\})\{\vec{x}' \mapsto \vec{x}\}$, where ϕ is a formula over the variables $\vec{x}, \vec{x}'$, ψ is over $\vec{x}$, and $\{\vec{x}' \mapsto \vec{x}\}$ is shorthand for the substitution $\{x'_1 \mapsto x_1, \ldots, x'_n \mapsto x_n\}$. The fixed point in the iteration is reached when $R_{i+1} \Leftrightarrow R_i$. Symbolic model checkers use representations of propositional formulas such as reduced ordered binary decision diagrams [Bryant 1992] for representing and computing images and fixed points efficiently and compactly. The image computation can also be done by a variant of satisfiability that generates the representation of all satisfying solutions. A different use of bounded model checking based on the construction of interpolants has proved to be quite effective in building over-approximations of the reachable state space [McMillan 2003]. Different approaches to symbolic model checking based on propositional satisfiability are surveyed and compared by Amla et al. [2005].

Bounded model checking can also be used for generating test cases corresponding to paths through the control flow graph [Hamon et al. 2005; Godefroid et al. 2005]. This is done by checking the satisfiability of the conjunction of conditions and transitions corresponding to the path. If it is satisfiable, a test case is generated as a satisfying assignment. If not, we know that the path is infeasible. The same approach can also be used to modify an existing test case to direct the computation along a different symbolic path.

Propositional logic is useful for model finding over a bounded universe. The Alloy language and system translate the problem of model finding in a first-order logic over relations to propositional satisfiability [Jackson 2006]. In Alloy, the variables range over m-ary relations over a bounded universe of cardinality n. The term language allows relations to be constructed using operations such as union, intersection, transposition, join, comprehension, and transitive closure. The basic predicates over these relational terms are those of subset, emptiness, and singularity. As we saw with the pigeonhole example, an m-ary relation over a universe of cardinality n can be represented by n^m propositional atoms and constants. The relational operations are defined as matrix operations. The resulting formulas are then translated to Boolean form for satisfiability checking. Many problems over sets and relations exhibit symmetry. It is therefore enough to look for just one model in each equivalence class given by a partitioning of the universe with respect to symmetry. By searching for models over a finite universe, Alloy is able to detect the presence of bugs and anomalies in specifications and programs [Torlak and Jackson 2007].

2.2 First-Order Logic

In propositional logic, the propositions are treated as atomic expressions ranging over truth values. Thus the proposition "Mary has a book" is either true or false. First-order logic admits individual variables that range over objects such as "Mary", predicates such as $Book(x)$ that represents the claim that the variable x is a book, and relations such as $Has(y, x)$ that expresses the claim that person y has object x. It also has existential quantification to represent, for example, the proposition "Mary has a book" as $\exists x.Book(x) \wedge Has(Mary, x)$. Universal quantification can be used to express the claim "Mary has only books" as $\forall x.Has(Mary, x) \Rightarrow Book(x)$. First-order logic also has function symbols that can be used to assert, for example, that "Mary's father has a book", by writing $\exists x.Book(x) \wedge Has(father(Mary), x)$.

The equality relation has a special role in first-order logic. It can be treated as a relation that satisfies certain axioms, or it can be treated as a logical symbol with a fixed interpretation. We take the latter approach and present first-order logic as a series of increasingly expressive fragments. A first-order language is built from a signature $\Sigma[X]$ where Σ contains function and predicate symbols with associated arities and X is a set of variables.

The signature $\Sigma[X]$ can be used to construct terms and formulas, where x ranges over the variables in X, f ranges over the n-ary function symbols in Σ, and p ranges over the n-ary predicate symbols in Σ.

—*Terms* $t := x \mid f(t_1, \ldots, t_n)$

—*Formulas* $\psi := \; p(t_1, \ldots, t_n) \mid t_0 = t_1 \mid \neg\psi_0 \mid$
$\psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid (\exists x.\psi_0) \mid (\forall x.\psi_0)$

Given a first-order signature Σ (ignoring variables), a first-order Σ-structure M consists of

—A non-empty *domain* $|M|$

—A map $M(f)$ from $|M|^n \to |M|$, for each n-ary function symbol $f \in \Sigma$

—A map $M(p)$ from $|M|^n \to \{\top, \bot\}$, for each n-ary predicate symbol p.

For example, if $\Sigma = \{0, +, <\}$, then we can define a Σ-structure M such that $|M| = \{a, b, c\}$ $+$ interpreted as addition modulo 3, where a, b, and c represent 0, 1, and 2, respectively.

$$\begin{aligned} M(0) &= a \\ M(+) &= \left\{ \begin{array}{l} \langle a,a,a\rangle, \langle a,b,b\rangle, \langle a,c,c\rangle, \langle b,a,b\rangle, \langle c,a,c\rangle, \\ \langle b,b,c\rangle, \langle b,c,a\rangle, \langle c,b,a\rangle, \langle c,c,b\rangle \end{array} \right\} \\ M(<)(x,y) &= \left\{ \begin{array}{l} \top, \text{ if } \langle x,y\rangle \in \{\langle a,b\rangle, \langle b,c\rangle\} \\ \bot, \text{ otherwise.} \end{array} \right. \end{aligned}$$

A $\Sigma[X]$-structure M also maps variables in X to domain elements in $|M|$. The interpretation $M[\![s]\!]$ of a $\Sigma[X]$-term s as an element of $|M|$ is defined as

$$\begin{aligned} M[\![x]\!] &= M(x) \\ M[\![f(s_1, \ldots, s_n)]\!] &= M(f)(M[\![s_1]\!], \ldots, M[\![s_n]\!]) \end{aligned}$$

Given a $\Sigma[X]$-structure M, let $M[x \mapsto \mathbf{a}]$ be a structure that maps x to $\mathbf{a}$ but behaves like M, otherwise. The interpretation $M[\![\phi]\!]$ of a $\Sigma[X]$-formula ϕ in a $\Sigma[X]$-structure M is defined as

$$\begin{aligned} M \models s = t &\iff M[\![s]\!] = M[\![t]\!] \\ M \models p(s_1, \ldots, s_n) &\iff M(p)(\langle M[\![s_1]\!], \ldots, M[\![s_n]\!]\rangle) = \top \\ M \models \neg\psi &\iff M \not\models \psi \\ M \models \psi_0 \vee \psi_1 &\iff M \models \psi_0 \text{ or } M \models \psi_1 \\ M \models \psi_0 \wedge \psi_1 &\iff M \models \psi_0 \text{ and } M \models \psi_1 \\ M \models (\forall x.\psi) &\iff M[x \mapsto \mathbf{a}] \models \psi, \text{ for all } \mathbf{a} \in |M| \\ M \models (\exists x.\psi) &\iff M[x \mapsto \mathbf{a}] \models \psi, \text{ for some } \mathbf{a} \in |M| \end{aligned}$$

For example, the following claims hold of the Σ-structure given above, where $\Sigma = \{0, +, <\}$.

Reflexivity	$\dfrac{}{\Gamma \vdash a = a, \Delta}$
Symmetry	$\dfrac{\Gamma \vdash a = b, \Delta}{\Gamma \vdash b = a, \Delta}$
Transitivity	$\dfrac{\Gamma \vdash a = b, \Delta \qquad \Gamma \vdash b = c, \Delta}{\Gamma \vdash a = c, \Delta}$
Function Congruence	$\dfrac{\Gamma \vdash a_1 = b_1, \Delta \ldots \Gamma \vdash a_n = b_n, \Delta}{\Gamma \vdash f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n), \Delta}$
Predicate Congruence	$\dfrac{\Gamma, p(a_1, \ldots, a_n) \vdash a_1 = b_1, \Delta \ldots \Gamma, p(a_1, \ldots, a_n) \vdash a_n = b_n, \Delta}{\Gamma, p(a_1, \ldots, a_n) \vdash p(b_1, \ldots, b_n), \Delta}$

Fig. 3. Proof rules for equality

(1) $M \models (\forall x, y.(\exists z. +(y, z) = x))$.
(2) $M \not\models (\forall x.(\exists y.x < y))$.
(3) $M \models (\forall x.(\exists y. +(x, y) = x))$.

A $\Sigma[X]$-formula ϕ is *satisfiable* if there is a $\Sigma[X]$-structure M such that $M \models \phi$, i.e., M is a *model* for ϕ. Otherwise, the formula ϕ is *unsatisfiable*. The set of *free variables* in a formula ϕ is given by $vars(\phi)$. For example, the set of free variables of the formula $(\forall x.x < y) \vee (\exists y.x < y)$ is $\{x, y\}$. A formula with an empty set of free variables is a *sentence*. For a sequence of variables $x_1, \ldots, x_n$ abbreviated as $\overline{x}$, let $\exists\overline{x}.\phi$ represent $\exists x_1. \ldots \exists x_n.\phi$. If a formula ϕ is satisfiable, so is its existential closure $\exists\overline{x}.\phi$, where $\overline{x}$ is $vars(\phi)$. A $\Sigma[X]$-formula ϕ is *valid* if $M \models \phi$ in every $\Sigma[X]$-structure. If a formula ϕ is unsatisfiable, then the negation of its existential closure $\neg\exists\overline{x}.\phi$ is valid, e.g., $\neg(\forall x.(\exists y.x < y))$. Note that if $\phi \wedge \neg\psi$ is unsatisfiable, $\phi \Rightarrow \psi$ is valid.

We introduce the proof theory of first-order logic in a series of fragments. Propositional logic is the fragment of first-order logic where there are no terms and all predicate symbols are 0-ary. The proof rules from Figure 2 however apply not just to propositional logic but to the propositional skeleton of formulas where the atoms can be *atomic formulas* which include both atoms of the form $s = t$ and $p(t_1, \ldots, t_n)$ for an n-ary predicate p, as well as quantified formulas.

The first set of sequent calculus proof rules shown in Figure 3 introduces equality with rules for reflexivity, symmetry, transitivity, and congruence.

These rules in sequent form lack the symmetry of the inference rules for the propositional connectives. Reflexivity is presented as an axiom, i.e., a rule with no premises, whereas symmetry, transitivity, and function and predicate congruence are presented as proof rule schemas. There is a congruence rule for each application of a function or predicate symbol to a sequence of terms. These congruence rules can also be captured by the axiom scheme $a_1 = b_1, \ldots, a_n = b_n \vdash f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$, for each n-ary function symbol f, and the axiom scheme $a_1 = b_1, \ldots, a_n = b_n, p(a_1, \ldots, a_n) \vdash p(b_1, \ldots, b_n)$, for each n-ary predicate symbol p.

Equational Logic. This is a fragment of first-order logic restricted to equality judgments which are sequents of the form $E \vdash a = b$, where E is a set of equations, each of which is implicitly universally quantified [Burris and Sankappanavar 1981]. A natural deduction presentation of equational logic is given by the rules in Figure 4. A *substitution* σ maps

Axiom	$\frac{}{E \vdash (a = b)\sigma}$, for $a = b \in E$ and substitution σ
Reflexivity	$\frac{}{E \vdash a = a}$
Transitivity	$\frac{E \vdash a = b \qquad E \vdash b = c}{E \vdash a = c}$
Congruence	$\frac{E \vdash a_1 = b_1 \ldots E \vdash a_n = b_n}{E \vdash f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)}$

Fig. 4. Equational Logic

variables to terms so that $f(a_1, \ldots, a_n)\sigma = f(a_1\sigma, \ldots, a_n\sigma)$. Equational logic is sound and complete in the sense that $E \vdash a = b$ is derivable iff every model of E is a model of $a = b$. Many theories such as semigroups, monoids, groups, rings, and Boolean algebras can be formalized in equational logic. Term rewriting systems [Baader and Nipkow 1998] can be used to prove equalities in certain equational theories.

McCune's celebrated proof of the Robbins conjecture with the EQP theorem prover is an exercise in equational logic. Given a binary operator $+$ and a unary operator n such that $+$ is associative and commutative, a Robbins algebra satisfies Robbins's equation

$$n(n(x + y) + n(x + n(y))) = x,$$

whereas a Boolean algebra satisfies Huntington's equation

$$n(n(x) + y) + n(n(x) + n(y)) = x.$$

The question of whether Robbins's equation implied Huntington's equation remained open for sixty years until McCune [1997] in 1996 used EQP to show that Robbins's equation does imply Huntington's equation. The actual proof shows that Robbins's equation implies $\exists C.\exists D.n(C+D) = n(C)$ which was already known to imply Huntington's equation. The existential quantification suggests that this proof is beyond equational logic, but the actual proof demonstrates a specific C and D, that can be used to construct an equational proof (see `http://www.cs.unm.edu/~mccune/ADAM-2007/robbins/`).

First-Order Logic. The next step is to introduce quantification. We have already seen the limited use of quantification in the equational logic framework where the equations in E are implicitly universally quantified. The sequent proof rules for the universal and existential quantifiers are given in Figure 5. In any application of the $\forall$-right and $\exists$-left rules, the constant c that appears in the premise sequents must be chosen so that it does not appear in the conclusion sequent. With this proviso, it is easy to check that the inference rules are sound. If the premise of the $\forall$-left rule is valid because $M[\![A\{x \mapsto t\}]\!] = \bot$, then $M[\![\forall x.A]\!] = \bot$ also holds. In the $\forall$-right rule, if the conclusion is not valid because there is an M such that $M[\![\gamma]\!] = \top$ for each $\gamma \in \Gamma$, $M[\![\delta]\!] = \bot$ for each $\delta \in \Delta$, and $M[\![\forall x.A]\!] = \bot$, then for some a in $|M|$, $M[x \mapsto a][\![A]\!] = \bot$. In this case, the premise is also invalid because $M[c \mapsto a][\![A\{x \mapsto c\}]\!] = \bot$.

The first-order logic proof system in Figures 2, 3, and 5 is sound and complete. Soundness is easily established since each proof rule yields a valid conclusion when given valid premises. Let $\neg\Delta$ abbreviate the set $\{\neg\phi \mid \phi \in \Delta\}$. For completeness [Gödel 1930; Henkin 1949; 1996], we must show that whenever a sequent $\Gamma \vdash \Delta$ in a first-order sig-

	Left	Right
$\forall$	$\dfrac{\Gamma, A\{x \mapsto t\} \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta}$	$\dfrac{\Gamma \vdash A\{x \mapsto c\}, \Delta}{\Gamma \vdash \forall x.A, \Delta}$
$\exists$	$\dfrac{\Gamma, A\{x \mapsto c\} \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta}$	$\dfrac{\Gamma \vdash A\{x \mapsto t\}, \Delta}{\Gamma \vdash \exists x.A, \Delta}$

Fig. 5. Sequent proof rules for quantification. The constant c must be *fresh*: it must not occur in the conclusion sequents of the proof rules $\forall$-right and $\exists$-left.

nature Σ is not provable, then the set of sentences $\Gamma \cup \neg\Delta$ has a Σ-model. We show that any *consistent* set of sentences Γ, i.e., where $\Gamma \vdash \emptyset$ (with an empty consequent) is not provable, has a model. For this, we start with $\Sigma_0 = \Sigma$ and $\Gamma_0 = \Gamma$ and introduce a fresh constant c_ϕ for each existential Σ_0 sentence ϕ of the form $\exists x.\psi$ along with the Henkin axioms $(\exists x.\psi) \Rightarrow \psi\{x \mapsto c_\phi\}$ to obtain Σ_1 and Γ_1. Iterating the addition of constants so that $\Sigma_{i+1} = \Sigma_i \cup C_i$, where C_i is the set of constants c_ϕ for each Σ_i-sentence $\exists x.\psi$ that is not a Σ_{i-1}-sentence. Similarly, $\Gamma_{i+1} = \Gamma_i \cup \{(\exists x.\psi) \Rightarrow \psi\{x \mapsto c_\phi\} | c_\phi \in C_i\}$. Let $\hat{\Gamma}$ denote $\bigcup_i \Gamma_i$, the result of adding all the Henkin constants and axioms to Γ. As with the completeness proof in Section 2.1, page 7, we order the sentences in $\hat{\Gamma}$ into a sequence $\phi_0, \phi_1, \ldots$ and define Θ_i so that $\Theta_0 = \Gamma'$, and $\Theta_{i+1} = \Theta_i \cup \{\phi_i\}$ if $\Theta_i \not\vdash \neg\phi_i$, and $\Theta_{i+1} = \Theta_i \cup \{\neg\phi_i\}$, otherwise. It can be shown that the set Θ is consistent. Moreover, note that for any $\phi_1 \vee \phi_2$ in Θ, either ϕ_1 or ϕ_2 is in Θ. Also, whenever ϕ of the form $\exists x.\phi_1$ is Θ, then $\phi_1\{x \mapsto c_\phi\}$ is in Θ. Exactly one of ϕ or $\neg\phi$ is in Θ. This means that we can read off a model from Θ where the domain consists of the equivalence classes of the ground (i.e., variable-free) terms with respect to the equalities in Θ, and the ground atoms in Θ are all assigned true.

A sentence ϕ in first-order logic can be converted into the equivalent *prenex normal form*: a first-order formula of the form $Q_1 x_1. \cdots Q_n x_n.\psi$, where each Q_i is either a universal or an existential quantifier and ψ is a quantifier-free formula. This is done by applying equivalence-preserving transformations to move the quantifiers outwards so that $(Qx.\phi) \bowtie \psi$ becomes $(Qy.\phi\{x \mapsto y\} \bowtie \psi)$ for a variable y that does not occur free in ψ, where $\bowtie$ is either $\wedge$ or $\vee$, and Q is either $\forall$ or $\exists$. Similarly, $\neg\forall x.P$ becomes $\exists x.\neg P$, and $\neg\exists x.P$ becomes $\forall x.\neg P$. A prenex sentence can be placed in an equivalent *Skolemized* form by iteratively replacing the existential quantifiers in the formula by *Skolem functions* as follows. Let Q_i be the first existential quantifier in the prefix of the formula so that it is preceded by the universally quantified variables $x_1, \ldots, x_{i-1}$. We then transform $\forall x_1. \ldots \forall x_{i-1}.\exists x_i.\psi$ to $\forall x_1. \ldots \forall x_{i-1}.\psi\{x_i \mapsto f_i(x_1, \ldots, x_{i-1})\}$, where f_i is a freshly chosen (Skolem) function symbol. By successively eliminating existential quantifiers in this manner, we arrive at a formula of the form $\forall \hat{x}_1. \ldots \forall \hat{x}_m.\hat{\psi}$ that is equisatisfiable with ϕ, where $\hat{\psi}$ is a quantifier free formula and $\hat{x}_1, \ldots, \hat{x}_m$ are the universally quantified variables from $x_1, \ldots, x_n$.

The Herbrand theorem [1930] asserts that such a sentence $\forall \hat{x}_m.\hat{\psi}$ is unsatisfiable iff there some Herbrand expansion of the form $\psi_1 \wedge \ldots \wedge \psi_k$ that is unsatisfiable, where each ψ_i is of the form $\hat{\psi}\{x_1 \mapsto t_{i1}, \ldots, x_{n'} \mapsto t_{in'}\}$ for Herbrand terms t_{ij}. The Herbrand terms are those built from fresh variables and the function symbols occurring in $\hat{\psi}$.

First-order logic has a number of other interesting metatheoretic properties. A set of first-order sentences is satisfiable if every finite subset of it is (compactness). A countable set of sentences with an infinite model has a model of any infinite cardinality (Löwenhiem-Skolem theorem). The amalgamation theorem for first-order logic yields a way of constructing an *amalgamated* model from two compatible models over possibly overlapping signatures [Hodges 1997]. Meta-theorems like the Robinson's joint consistency theorem, the Craig interpolation theorem, and the Beth definability theorem are corollaries of the amalgamation theorem. Church [Church 1936] and Turing [Turing 1965] showed that the problem of deciding validity for first-order logic sentences is *undecidable*. The halting problem for Turing machines, which is easily seen to be unsolvable, can be expressed in first-order logic.

First-Order Theories. Given a signature Σ, a Σ-*theory* is a set of first-order sentences Γ closed under *entailment*, i.e., for any Σ-formula ϕ, if for all Σ-structures M such that $M \models \psi$ for each $\psi \in \Gamma$, it is the case that $M \models \phi$, then $\phi \in \Gamma$. A first-order Σ-theory can be equivalently given by a set of Σ-models $\mathcal{M}$ closed under isomorphism and variable reassignment, in which case it corresponds to the set Γ of first-order Σ-sentences that are valid in each of the models in $\mathcal{M}$. A theory Γ is *finitely axiomatizable* if the sentences in Γ are entailed by some finite subset of Γ. Many theories are finitely axiomatizable, and several of these are even equational. Theories that are not finitely axiomatizable can still be *recursively axiomatizable* so that the axioms can be distinguished from non-axioms by a program. A theory is complete if for each sentence ϕ, either ϕ or $\neg\phi$ is in the theory. A theory is *decidable* if there is a program that can distinguish sentences in the theory from those that are not. First-order theories can be shown to be decidable by directly presenting a decision procedure or by interpreting the theory within another theory that is known to be decidable. One typical way of showing decidability is by showing that it admits *quantifier elimination* so that any subformula $\forall x.\phi$ or $\exists x.\phi$ with a quantifier-free ϕ can be replaced by an equivalent quantifier-free formula $\hat{\phi}$. The *word problem* (WP) for a theory is that of checking if $\forall p$ is valid in the theory for an atom p. The *uniform word problem* (UWP) for a theory is that of determining if the sentence $\forall(\bigwedge \Gamma \Rightarrow p)$ is valid in the theory, for a finite set of atoms Γ and an atom p. The *clausal validity problem* (CVP) for a theory is that if $\forall(\bigvee \Gamma)$ is valid in the theory, where Γ is a finite set of atom and negations of atom. A solution to the CVP for a theory would also solve the corresponding UWP, and one for the UWP for a theory would also solve the WP.

We briefly enumerate a few interesting first-order theories.

—The theory of identity in the empty signature has no axioms and can be used to make assertions about the minimum or maximum number of distinct elements in a model. If a sentence ϕ with r variables is satisfiable, then it is satisfiable in a model that contains no more than r elements. This theory therefore has the *finite model property* and is hence decidable.

—The empty theory over Σ is essentially first-order logic without axioms. This theory is undecidable. Any finitely axiomatizable first-order theory T can be expressed in this fragment since the validity of ϕ in T is equivalent to the validity of $\Gamma \vdash \phi$ in the empty theory. Certain fragments of this theory are decidable. For example, the validity of a sentence with k monadic predicates and r variables is decidable by checking with models of size at most $2^k r$, but adding even a single two-place predicate renders the fragment undecidable [Boolos and Jeffrey 1989]. The Bernays-Schönfinkel fragment is of the form $\exists x_1, \ldots, x_m.\forall y_1, \ldots, y_n \phi$ where ϕ is a quantifier-free formula. The validity of such a formula is decidable if given a set of m distinct fresh constants $c_1, \ldots, c_n$, and a sequence $v_1, \ldots, v_n$ of these constants, the formula $\phi\{x_1 \mapsto c_1, \ldots, x_m \mapsto c_m, y_1 \mapsto v_1, \ldots, y_n \mapsto v_n\}$ is in a decidable fragment. Several decidable and undecidable fragments of first-order logic are covered by Börger, Gurevich, and Grädel [1997].

—The theory of a single relation such as an equivalence or an ordering relation can be easily given by adding axioms for properties such as reflexivity, transitivity, symmetry or anti-symmetry as axioms, as well as those for linearity, and dense or discrete orders. These theories are all decidable [Rabin 1978].

—Algebraic theories such as those for semigroups, groups, rings, fields, and Boolean algebras can be axiomatized through the basic algebraic laws. The theory of *algebraically closed fields* (ACF) is a field where every univariate polynomial of nonzero degree has a root. The theory of *real closed fields* is an ordered field where each non-negative element has a square root, and each odd degree polynomial has a root. The theory of torsion-free groups where $\forall x.x^n = 1 \Rightarrow x = 1$ for any n, is an example that is not finite axiomatizable, but is recursively axiomatizable. The first-order theories of BA, ACF and RCF are decidable, whereas that of groups is undecidable. Even the uniform word problem for semigroups, groups, and rings is undecidable, whereas the UWP for commutative semigroups and groups is decidable.

—Various fragments of arithmetic are common examples of first-order theories [Bockmayr and Weispfenning 2001]. The theory of linear arithmetic over the signature $\langle 0, 1, + \rangle$, also known as Presburger arithmetic, is decidable by means of quantifier elimination in triple-exponential time [Cooper 1972; Fischer and Rabin 1974]. The clausal validity problem for linear arithmetic equality and inequality constraints over integers is NP-complete, whereas it can be solved in polynomial time over the reals and rationals. However, the theory of natural numbers, integers, and rational numbers with multiplication (described below) is both incomplete and undecidable. Even the fragment that is restricted to solving polynomial equalities is undecidable, thus yield a negative answer to Hilbert's tenth problem [Matiyasevich 1993]. However, the first-order theory of addition and multiplication over the real numbers is decidable by means of quantifier elimination in doubly exponential time [Basu et al. 2003].

—Useful theories for various datatypes can often be captured by equational axioms. For example, a simple theory of lists is given by the axioms:

(1) $car(cons(x, y)) = x$

(2) $cdr(cons(x, y)) = y$

—Other theories need to make use of the logical connectives, as with the theory of non-extensional arrays below.

(1) $select(update(a, i, v), i) = v$

(2) $i \neq j \Rightarrow select(update(a, i, v), j) = select(a, j)$

The list and array theories above are missing extensionality axioms that can be used to show that two lists or two arrays are equal if they share the same elements. For lists, the extensionality axiom can be written as $cons(x, y) = cons(u, v) \Rightarrow (x = u \wedge y = v)$. However, for arrays, we need to use quantification to assert that $(\forall i.select(a, i) = select(b, i)) \Rightarrow a = b$. The boundary between decidability and undecidability for the theory of arrays is explored by Bradley, Manna, and Sipma [2006]. The clausal validity

(1) $S(x) \neq 0$
(2) $S(x) = S(y) \Rightarrow x = y$
(3) $x = 0 \vee (\exists y.x = S(y))$
(4) $x + 0 = x$
(5) $x + S(y) = S(x + y)$
(6) $x \times 0 = 0$
(7) $x \times S(y) = (x \times y) + x$
(8) An axiom scheme such that for each formula ϕ with $vars(\phi) = \{x\}$,

$$\phi\{x \mapsto 0\} \wedge (\forall x.\phi \Rightarrow \phi\{x \mapsto S(x)\}) \Rightarrow (\forall x.\phi)$$

Fig. 6. Axioms for a first-order theory of arithmetic

problem for the extensional theory of arrays is NP-complete [Downey and Sethi 1978; Stump et al. 2001] (see page 31).

An axiomatization of arithmetic was first formulated by Dedekind and Peano but their presentation makes use of quantification over sets. The first-order theory of arithmetic in the signature $\langle 0, 1, +, \times \rangle$ shown in Figure 6 employs the unary successor operation S and the binary operations $+$ and $\times$, and a binary ordering predicate $\leq$. Without the induction axiom scheme, the system is called Robinson arithmetic and is sufficient for demonstrating the incompleteness of arithmetic [Tarski et al. 1971].

An alternate way to define the *standard* first-order theory of arithmetic would be as the set of sentences that are valid in the standard arithmetic interpretation of 0, 1, $+$, and $*$ over the natural numbers. In this interpretation, the theory contains every sentence or its negation. However, the theory obtained from the axiomatization in Figure 6 does not coincide with the standard theory. This axiomatization of arithmetic is *incomplete* as demonstrated by Gödel [1967]: either the axioms are inconsistent or there are *undecidable* sentences that are neither provable nor disprovable. Indeed, the problem persists for any recursive first-order axiomatization of arithmetic, or even a *recursively enumerable* one, i.e., where the axioms can be listed by a program even if they cannot be recognized by one. The set of sentences in the standard theory, assuming consistency, is not recursively enumerable. Gödel's second incompleteness theorems demonstrates that the consistency of the theory of arithmetic, the arithmetical statement that $0 = 1$ is unprovable, is itself an undecidable sentence. The second incompleteness theorem effectively defeats Hilbert's programme of establishing the consistency of formalized mathematics by finitistic methods. This was the second of his twenty-three problems presented at the International Congress of Mathematics in 1900 [Hilbert 1902].

Barwise [1978b] is an excellent introduction to first-order logic. Hodges [1997] contains a readable introduction to model theory. Proof systems for first-order logic are covered by Kleene [1952], Gentzen [1969], and Smullyan [1968].

Primitive recursive arithmetic (PRA) was introduced by Skolem [1967] and Goodstein [1964] to provide a finitist foundation for mathematics. In addition to the basic constant, successor, and projection operations, the theory allows new functions to be defined in terms of old ones by the schemas of composition and primitive recursion. A definition by composition has the form

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)),$$

where the new function f is defined in terms of existing functions $g, h_1, \ldots, h_m$. A definition by primitive recursion has the form

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= b(x_1, \ldots, x_n) \\ f(S(x), x_1, \ldots, x_n) &= h(f(x, x_1, \ldots, x_n), x, x_1, \ldots, x_n) \end{aligned}$$

Goodstein's rule of Recursion-Induction [Goodstein 1964] asserts that two expressions that satisfy the same primitive recursion scheme are equivalent. This rule was independently proposed by McCarthy [1963] in the context of his theory of pure Lisp. McCarthy's formalization of Lisp is also the foundation for the Boyer–Moore family of interactive theorem provers [Boyer and Moore 1979; 1988; Kaufmann et al. 2000].

Set Theory. Many concepts in mathematics such as structures, orders, and maps can be encoded using sets. The original conception of a set as a collection of all elements satisfying a property was found to be unsound. For example, Russell's paradox introduces the set of elements R consisting of all elements that do not belong to themselves, so that $R \in R \iff R \notin R$. Similarly, the Burali-Forti paradox constructs the set Ω of all ordinal numbers so that Ω is both an ordinal number and is larger than all the ordinal numbers. These paradoxes drove the first-order formalization of set theory by Zermelo, Fraenkel, and Skolem. In ZF set theory, there is one basic predicate $\in$ denoting set membership. Sets are constructed by means of *pairing*, *union*, *infinity*, *power set*, *comprehension*, and *replacement*. Comprehension is restricted to defining as a set, a subset $\{x \in A | \phi(x)\}$ of a given set A satisfying a stated property $\phi(x)$. The replacement axiom defines as a set, the image of a set with respect to a map specified by a formula. The axiom of *regularity* or *foundation* asserts that each set x contains an element y that has no elements in common with x. The axiom of *extensionality* asserts that two sets are equal if all their elements are in common, so that sets are completely characterized by their members.

Set theory is used in various systems for specification and verification such as Z [Abrial 1980; Spivey 1993], B [Abrial 1996], Z/Eves [Saaltink 1997], and Isabelle/ZF [Paulson 2003]. First-order logic theorem provers have been successful in formalizing proofs in set theory [Boyer et al. 1986; Quaife 1992; Belinfante 1999] by exploiting the finitely axiomatizable set theory due to von Neumann, Bernays, and Gödel [Mendelson 1964]. Good introductions to set theory include Skolem [1962], Halmos [1960], Suppes [1972], and Kunen [1980].

Higher-Order Logic. In first-order logic, the variables range over individuals whereas the function and predicate symbols are treated as constants. A first-order formula such as $(\forall x.p(x, f(x))) \Rightarrow \forall x.\exists y.p(x, y)$ is valid if it is true in all interpretations of the function and predicate symbols. Higher-order logic allows quantification over function and predicate symbols. Second-order logic admits quantification over first-order function and predicate symbols: for example, second-order logic can assert $\exists f.\forall x.p(x, f(x)) \Rightarrow \forall x.\exists y.p(x, y)$. Second-order logic can be used to define the concept of injective and surjective functions, formalize finiteness, define inductive predicates like transitive closure and reachability, and formalize recursive datatypes like the natural numbers, lists, and trees. Third-order logic admits quantification over functions that take first-order functions as arguments, and predicates that take first-order functions and predicates as arguments. Higher-order logic includes n-th order logic for any natural number $n \geq 1$.

Historically, typed higher-order logic was used to counter the inconsistency in Frege's

Grundgesetze [Frege 1903] system of logic which admits a form of unrestricted comprehension as a way of defining sets. With this, it is easy to derive *Russell's paradox*. Zermelo [1908] avoided the contradiction by restricting the range of comprehension to an existing set, as in $\{x \in y \mid \phi\}$. Russell [1903; 1908] developed a hierarchical type system with individuals, propositions, predicates over individuals, and predicates of predicates, and so on.

Church's simple theory of types [Church 1940] starts with two basic types of individuals i and propositions o at level 0 and builds a hierarchy of functions such that if S is a type at level n and T is a type at level $n+1$, then $S{\rightarrow}T$ is a type at level $n+1$.

$$T := i \mid o \mid T_1{\rightarrow}T_2.$$

The type hierarchy rules out the self-membership or self-application needed to define the Russell set. The terms of higher-order logic are defined from the basic constants of the type i and o using lambda-abstraction $\lambda(x : S).a$ and application $a\ b$.

$$s := x \mid \lambda(x : T).s \mid s_1\ s_2.$$

Terms are typed relative to a context Ξ of the form $x_1 : T_1, \ldots, x_n : T_n$, where $x_j \not\equiv x_k$ for $j \neq k$ and each T_i is a type. *Typing judgments* have the form $\Xi \vdash s : T$ for context Ξ, term s, and type T. The typing rules allow $\Xi \vdash \lambda(x : S).a : S{\rightarrow}T$ to be derived from $\Xi, x : S \vdash a : T$, and $\Xi \vdash (s\ t) : T$ to be derived from $\Xi \vdash s : S{\rightarrow}T$ and $\Xi \vdash t : S$. These typing rules are the same as the proof rules for natural deduction given by the Curry–Howard isomorphism in Figure 13.

There are many different ways to axiomatize higher-order logic depending on which primitives are assumed. Andrews [1986; 1940] presents a system Q_0 with equality as primitive so that we have $\Leftrightarrow$: $o{\rightarrow}(o{\rightarrow}o)$ and $=$: $S{\rightarrow}(S{\rightarrow}o)$ for each type S. With this, we can define the truth value $\top$ as $= (\Leftrightarrow)(\Leftrightarrow)$. We revert to the infix notation for familiar symbols so that we write $\top$ as $\Leftrightarrow = \Leftrightarrow$. With this, we can define the everywhere-$\top$ function over some type T as $\lambda(x : T).\top$. With this, $\bot$ abbreviates $\lambda(x : o).x = \lambda(x : o).\top$, and universal quantification $\forall$ for a predicate P of type $T{\rightarrow}o$ can be defined so that $\forall p$ abbreviates $p = \lambda(x : T).\top$. A further abuse of notation abbreviates $\forall\lambda(x : T).a$ as $\forall(x : T).a$. The negation operator $\neg$ can be defined as $\lambda(x : o).(x = \bot)$. The conjunction operator $\wedge$ can be defined as $\lambda(x : o).\lambda(y : o).\forall(p : o{\rightarrow}(o{\rightarrow}o)).p(x)(y) = p(\top)(\top)$. From these primitives, it is easy to define implication and existential quantification.

A *term context* is a λ-term with a single occurrence of a hole $\{\}$ in it.

$$s^0 := \{\} \mid \lambda(x : T).s^0 \mid s^0\ s \mid s\ s^0.$$

We write $a\{\}$ to represent a term context so that $a\{b\}$ is the term that results from filling the hole in $a\{\}$ with b. Note that free occurrences of variables in b can become bound in $a\{b\}$.

The system Q_0 is a Hilbert-style system with one rule of inference

$$\frac{c\{a\} \qquad a = b}{c\{b\}}\ .$$

It has four axioms:

(1) $\vdash (g\ \top \wedge g\ \bot) = \forall(x : o).g\ x$, for $g : o{\rightarrow}o$

(2) $\vdash x = y \Rightarrow (g\ x \Leftrightarrow g\ y)$, for $g : T{\rightarrow}o$ and $x : T, y : T$

(3) $(f = g) \Leftrightarrow \forall(x : S).f\ x = g\ x$, for $f : S{\rightarrow}T$, $g : S{\rightarrow}T$
(4) $(\lambda(x : S).a)\ b = a\{x \mapsto b\}$, where no free occurrences of variables in b appear bound in $a\{x \mapsto b\}$.

The first axiom asserts that $\top$ and $\bot$ are the sole elements of o. The second axiom is the usual congruence rule for equality. The equivalence asserted in the third axiom can be read as a congruence rule in one direction, and an extensionality principle in the other. The last axiom introduces the equality between a redex term $\lambda(x : S).s\ t$ and its β-reduct $s\{x \mapsto t\}$. Two λ-terms are α-equivalent if one term can be obtained from the other one by uniformly renaming bound variables. Thus $\lambda(x : S).x\ y$ is α-equivalent to $\lambda(z : S).z\ y$. Such α-equivalent terms are treated as being syntactically interchangeable in the proof system.

Additionally, higher-order logic also has axioms asserting the existence of an infinite set and a choice operator $choose(P)$ such that $\exists x.P(x) \Rightarrow P(choose(P))$. These axioms correspond to the axioms of infinity and choice in set theory. Lambda-abstraction is similar to the comprehension principle for defining new sets.

Since higher-order logic can express finiteness, it does not satisfy compactness. The induction axiom scheme can be expressed as a single axiom, which makes it possible to define the natural numbers and other recursive datatypes directly in second-order logic. The μ-calculus [Park 1976] which extends first-order logic with least and greatest fixed points is also definable in higher-order logic. Many verification systems are based on higher-order logic since it is both simple and expressive [Gordon 1986].

In the above *standard* interpretation, even second-order logic is incomplete by Gödel's incompleteness theorem. However, higher-order logic is complete for *Henkin models* where the function type $T_1{\rightarrow}T_2$ is interpreted as any set of maps from $M[\![T_1]\!]$ to $M[\![T_2]\!]$ that includes as elements interpretations $M[\![\lambda(x : T_1).s]\!]$ for any lambda-abstraction $\lambda(x : T_1).s$ of type $T_1{\rightarrow}T_2$ [Henkin 1950; 1996].

Monadic second-order logic where the higher-order variables are restricted to monadic predicates is decidable. Automata-theoretic techniques can be used to show that monadic second-order logic remains decidable even with finitely many unary injective (successor) operations over the base type (SnS). The *weak* version, WSnS, where the monadic predicates are restricted to ranging over finite sets, is also decidable [Rabin 1978; Elgaard et al. 1998]. There are several excellent expositions of higher-order logic including Feferman [1978], Leivant [1994], and van Benthem and Doets [1983].

3. SATISFIABILITY SOLVERS

Satisfiability is a core technology for many verification tasks. We briefly survey the tools for propositional satisfiability and satisfiability modulo theories and their use in verification.

3.1 Inference Systems

Decision procedures for satisfiability are used to determine if a given formula has a model. If the procedure fails to find a model, it must be because the original formula is unsatisfiable. *Inference systems* [Shankar and Rueß 2002; Shankar 2005; de Moura et al. 2007] provide a unifying framework for defining such satisfiability procedures.

An inference system is a triple $\langle \Psi, \Lambda, \rhd \rangle$ consisting of a set Ψ of inference states, a mapping Λ from inference states to formulas, and a binary inference relation $\rhd$ between

Res	$\dfrac{K, k \vee \kappa_1, \overline{k} \vee \kappa_2}{K, k \vee \kappa_1, \overline{k} \vee \kappa_2, \kappa_1 \vee \kappa_2}$ $\begin{array}{l} \kappa_1 \vee \kappa_2 \notin K \\ \kappa_1 \vee \kappa_2 \text{ is not tautological} \end{array}$
Contrad	$\dfrac{K}{\bot}$ if $p, \neg p \in K$ for some p

Fig. 7. Inference System for Ordered Resolution

inference states. For each formula ϕ, there must be at least one state ψ such that $\Lambda(\psi) = \phi$. There is a special unsatisfiable inference state $\bot$. The inference relation $\rhd$ must be

(1) **Conservative:** If $\psi \rhd \psi'$, then $\Lambda(\psi)$ and $\Lambda(\psi')$ must be equisatisfiable.
(2) **Progressive:** For any subset S of Ψ, there is a state $\psi \in S$ such that there is no $\psi' \in S$ where $\psi \rhd \psi'$.
(3) **Canonizing:** If $\psi \in \Psi$ is irreducible, that is, there is no ψ' such that $\psi \rhd \psi'$, then either $\psi \equiv \bot$ or $\Lambda(\psi)$ is satisfiable.

We say that a function f is an *inference operator* when $\psi \rhd f(\psi)$ if there is a ψ' such that $\psi \rhd \psi'$, and otherwise, $f(\psi) = \psi$. Given an inference operator f, let $f^*(\psi) = f^i(\psi)$, for the least i such that $f^{i+1}(\psi) = f^i(\psi)$. Since $\vdash$ is progressive, such an i must exist. We can use the operation f^* as a decision procedure since ψ is unsatisfiable iff $f^*(\psi) = \bot$.

As an example, we present an inference system for *ordered resolution* in propositional logic as an illustration. The problem is to determine the satisfiability of a set K of input clauses. This set K also happens to be the input inference state. We assume that K does not contain any clause that is a tautology, i.e., one that contains both a literal and its negation. It is also assumed that duplicate literals within a clause are merged. We are also given an ordering $p \succ q$ on atoms, which can be lifted to literals as $\neg p \succ p \succ \neg q \succ q$. Clauses are maintained in decreasing order with respect to this ordering. In *binary resolution*, a clause κ containing k and a clause κ' containing $\overline{k}$, the complement of k, are resolved to yield $(\kappa - \{k\}) \cup (\kappa' - \{\overline{k}\})$, but in ordered resolution, only the maximal literals in a clause can be resolved. The inference system for ordered resolution is shown in Figure 7. The resolution rule **Res** adds the clause $\kappa_1 \vee \kappa_2$ obtained by resolving the clauses $k \vee \kappa_1$ and $\overline{k} \vee \kappa_2$ to K, provided $\kappa_1 \vee \kappa_2$ is not a tautology and is not already in K, and k and $\overline{k}$ are the maximal literals in $k \vee \kappa_1$ and $\overline{k} \vee \kappa_2$, respectively.

The resolution inference system can be applied to the example $\neg p \vee \neg q \vee r,\ \neg p \vee q,\ p \vee r,\ \neg r$ to achieve a refutation as shown below.

$$\begin{array}{cl} (K_0 =)\ \neg p \vee \neg q \vee r,\ \neg p \vee q,\ p \vee r,\ \neg r & \\ \hline & \textbf{Res} \\ (K_1 =)\ \neg q \vee r,\ K_0 & \\ \hline & \textbf{Res} \\ (K_2 =)\ q \vee r,\ K_1 & \\ \hline & \textbf{Res} \\ (K_3 =)\ r,\ K_2 & \\ \hline & \textbf{Contrad} \\ \bot & \end{array}$$

The correctness of the inference system is interesting since the resolution rule has been restricted to resolving only on maximal literals. The inference system is progressive since the input K_0 has a bounded number of atoms and every clause in K is constructed from these atoms. For n atoms, there are at most 3^n clauses that can appear in K. Since each resolution step generates at least one new clause, this bounds the size of the derivations. The inference system is conservative since any model M of $k \vee \kappa_1$ and $\overline{k} \vee \kappa_2$ is also a model of $\kappa_1 \vee \kappa_2$. Conversely, if K' is derived from K' by a resolution step, then $K \subseteq K'$, and hence any model of K' is also a model of K. Finally, the inference system is canonizing. Given an irreducible non-$\bot$ configuration K in the atoms $p_1, \dots, p_n$ with $p_i \prec p_{i+1}$ for $1 \leq i \leq n$, build a series of partial interpretations M_i as follows:

(1) Let $M_0 = \emptyset$.

(2) If p_{i+1} is the maximal literal in a clause $p_{i+1} \vee \kappa \in K$ and $M_i \not\models \kappa$, then let $M_{i+1} = M_i[p_{i+1} \mapsto \top]$.
Otherwise, let $M_{i+1} = M_i[p_{i+1} \mapsto \bot]$.

Each M_i satisfies all the clauses over just the atoms p_j for $j \leq i$, and hence $M = M_n$ satisfies K. Many inference procedures can be presented and analyzed as inference systems.

3.2 The DPLL procedure for Propositional Satisfiability

Given a propositional formula ϕ, checking whether there is an M such that $M \models \phi$ is a basic problem that has many applications. For simplicity, the formula is first transformed into CNF so that we are checking the satisfiability of a set of clauses K. One easy solution is to enumerate and check all possible assignments of truth values to the propositional atoms in K. We can systematically scan the space of assignments while backtracking to try a different assignment each time a branch of the search tree is found to contain no feasible assignments. This approach has two sources of redundancy. Truth assignments to some variables are implied by those of other variables. For example, if p is assigned $\bot$ and there is a clause $p \vee q$ in K, then clearly q must be assigned $\top$ and there is no need to pursue the branch where q is assigned $\bot$. A partial assignment triggers a conflict when, for example, there is a clause $p \vee \neg q$ where p is assigned $\bot$ and q is assigned $\top$. Typically, only a small subset of the partial assignment is needed to trigger such a conflict. Even when the other assignments are varied during the search, the same conflict is going to be triggered. The Davis–Putnam–Logemann–Loveland procedure [Davis and Putnam 1960; Davis et al. 1962] introduced the basic idea of searching for a satisfiable assignment by *branching* on the truth assignment to a variable, *propagating* any implied truth values, and *backtracking* from failed assignments. Building on the ideas in SATO [Zhang 1997] and GRASP [Marques-Silva and Sakallah 1999], modern satisfiability solvers such as Chaff [Moskewicz et al. 2001], zChaff [Zhang and Malik 2002], BerkMin [Goldberg and Novikov 2007], MiniSat [Eén and Sörensson 2003], Siege [Ryan 2004], and PicoSAT [Biere 2008] add very efficient *Boolean constraint propagation* to handle the first redundancy, and *conflict-directed backjumping* to eliminate the second source of redundancy.

The DPLL inference system looks for a satisfying assignment for a set of n clauses K over m propositional variables [Zhang and Malik 2003; Nieuwenhuis et al. 2006; de Moura et al. 2007]. It does this by building a *partial assignment* M in levels l and a set of implied *conflict clauses* C. A partial assignment M up to level l has the form $M_0; M_1; \dots; M_l$. The partial assignment M_0 is a set of pairs $k_i[\gamma_i]$ with literal k_i and *source clause* $\gamma_i \in K \cup C$. For $0 < i \leq l$, each M_i has the form $d_i : k_1[\gamma_1], \dots, k_n[\gamma_n]$ with decision literal d_i and *implied literals* k_j and their corresponding source clause γ_j. When k occurs as an implied literal in M, let $M_{<k}$ be the prefix of the partial assignment preceding the occurrence of k in M. We maintain the invariant that the source clause for an implied literal k in M can be written as $k \vee \gamma$ where $M_{<k} \models \neg\gamma$. We can view M as a partial assignment since

$M(p) = \top$ if p occurs in M, $M(p) = \bot$ if $\neg p$ occurs in M, and $M(p)$ is undefined, otherwise.

The inference state set Ψ consists of all 4-tuples of the form $\langle l, M, K, C\rangle$ containing the decision level l, the partial assignment M, the input clause set K that remains fixed, and the conflict clause set C. The operation $\Lambda(\langle l, M, K, C\rangle)$ returns $\bigwedge(M_0 \cup K \cup C)$. The DPLL inference system consists of four basic components

(1) *Propagation* is used to add all the implied literals k to the partial assignment M at the current decision level l. A literal k is implied if there is a clause $k \vee \gamma$ in $K \cup C$ where $M \models \neg\gamma$. Propagation can also detect an inconsistency when there is a clause γ in $K \cup C$ where $M \models \neg\gamma$. If this inconsistency is detected at decision level 0, then this reflects a contradiction in K since the clauses in C are implied by those in K.

(2) *Analysis* is applied when propagation detects an inconsistency that is not at level 0 and it constructs a *conflict clause* γ such that $M \models \neg\gamma$ and γ contains exactly one literal at the current level l. When propagation detects an inconsistency, there is a clause γ in $K \cup C$ such that $M \models \neg\gamma$. This clause can contain one or more literals that are falsified by M at the current level l. If we have just one such literal, then γ can be taken as a conflict clause. If we have more than one such literal, then one of these must be maximal in terms of the position of its assignment in M. We replace γ by the result of resolving γ with the source clause $\overline{k} \vee \gamma'$ for this maximal literal k. Since $M_{<\overline{k}} \models \neg\gamma$, we know that this resolution step only replaces k with literals whose negations precede $\overline{k}$ in M, so that the new clause γ is still falsified by M and has a smaller maximal literal. Since there are only a bounded number of literals at level l, we will eventually terminate with a *conflict clause* that has exactly one literal at level l.

(3) *Backjumping* is used to reset the partial assignment based on the conflict clause γ constructed by analysis. We know that γ is of the form $k \vee \gamma'$, where k is falsified at level l and γ' is falsified at some level $l' < l$. Let $M^{l'}$ represent the restriction of M to the assignments in levels at or below l'. The backjumping step replaces M with the partial assignment $M^{l'}, \overline{k}[\gamma]$ while adding γ to the conflict clause set C.

(4) *Selection* is employed to pick an unassigned literal k as the decision literal for continuing the search at the next level when propagation has been applied to extract all the implied literals at the current level l and no conflicts have been detected. The partial assignment M is then replaced by $M; k$. Note that when there are no more unassigned literals, the partial assignment M is a total assignment, and since it yields no conflict, we have $M \models \gamma$ for each clause γ in $K \cup C$.

An example of the procedure is shown in Figure 8. The given input clause set K is $\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$. Since there are no unit (single literal) clauses, there is no implied literal at level 0. We therefore select an unassigned literal, in this case s as the decision literal at level 1. Again, there are no implied literals at level 1, and we select an unassigned literal r as the decision literal at level 2. Now, we can add the implied literals $\neg q$ from the input clause $\neg q \vee \neg r$ and p from the input clause $p \vee q$. At this point, propagation identifies a conflict where the partial assignment M falsifies the input clause $\neg p \vee q$. The conflict is analyzed by replacing $\neg p$ with q to get the unit clause q. Since the maximal level of the empty clause is 0, backjumping yields a partial assignment q at level 0 while adding the unit clause q to the

step	l	M	K	C	γ
select s	1	$;s$	K	$\emptyset$	–
select r	2	$;s;r$	K	$\emptyset$	–
propagate	2	$;s;r,\neg q[\neg q \vee \neg r]$	K	$\emptyset$	–
propagate	2	$;s;r:\neg q, p[p \vee q]$	K	$\emptyset$	–
conflict	2	$;s;r:\neg q, p$	K	$\emptyset$	$\neg p \vee q$
analyze	0	$\emptyset$	K	q	–
backjump	0	$q[q]$	K	q	–
propagate	0	$q, p[p \vee \neg q]$	K	q	–
propagate	0	$q, p, r[\neg p \vee r]$	K	q	–
conflict	0	q, p, r	K	q	$\neg q \vee \neg r$

Fig. 8. The DPLL procedure with input $\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$

conflict clause set C. Propagation then yields the implied literals p from the input clause $p \vee \neg q$ and r from the input clause $\neg p \vee r$, which leads to the falsification of the input clause $\neg q \vee \neg r$. Since this conflict occurs at level 0, we report unsatisfiability.

Generating Proofs. The DPLL search procedure can be augmented to generate proofs by annotating the conflict clauses with proofs corresponding to the analysis steps used in generating them [Zhang and Malik 2003]. In the example above, the conflict q can be annotated with the proof $resolve(p, \neg p \vee q, p \vee q)$ to indicate that the clause is generated by resolving $\neg p \vee q$ and $p \vee q$ on the atom p. The final conflict clause $\neg q \vee \neg r$ can also be analyzed to construct the proof shown below. The conflict clause q is used as an input here, but its proof computed during analysis can be spliced into this proof.

$$\dfrac{\dfrac{\dfrac{\neg q \vee \neg r \qquad \neg p \vee r}{\neg q \vee \neg p} \qquad p \vee \neg q}{\neg q} \qquad q}{\bot}$$

Generating Interpolants. The Craig Interpolation Lemma [Craig 1957] states that if we have two sets of first-order logic formulas Γ and Δ such that $\Gamma \cup \Delta$ is inconsistent, then there is a formula ϕ in the intersection of the function and predicate symbols from Γ and Δ such that Γ entails ϕ and Δ entails $\neg\phi$. For sets of propositional formulas Γ and Δ, the interpolant ϕ is a propositional formula whose atoms appear in both Γ and Δ. Interpolants are useful for finding useful program assertions including invariants [McMillan 2003]. For example, we already saw with bounded model checking that when the assertion

$$I(\vec{x}_0) \wedge \bigwedge_{i=0}^{k-1} N(\vec{x}_i, \vec{x}_{i+1}) \wedge (\bigvee_{j=0}^{k} \neg P(\vec{x}_j))$$

is unsatisfiable, we do not have any violations of property P in the first k steps of the computation. An interpolant can be constructed from this proof of unsatisfiability where Γ consists of $I(\vec{x}_0) \wedge N(\vec{x}_0, \vec{x}_1) \wedge (\neg P(\vec{x}_0) \vee \neg P(\vec{x}_1))$ and Δ consists of $\bigwedge_{i=1}^{k-1} N(\vec{x}_i, \vec{x}_{i+1}) \wedge (\bigvee_{j=2}^{k} \neg P(\vec{x}_j))$. Now Γ and Δ only overlap on $\vec{x}_1$ so that their interpolant yields an assertion on $\vec{x}_1$ that conservatively over-approximates the image of the initial state with respect to the transition relation. The interpolant can be used as the initial state in the next iteration of bounded model checking. We examine the construction of an interpolant from a refutational proof based on resolution.

Let the input clause set K be partitioned into K_1 with atoms $atoms(K_1)$ and K_2 with atoms $atoms(K_2)$. We show that if K is unsatisfiable, there is a formula (an interpolant) I such that $K_1 \Rightarrow I$ and $K_2 \wedge I \Rightarrow \bot$. Furthermore, $atoms(I) \subseteq atoms(K_1) \cap atoms(K_2)$.

An interpolant I_Γ can be constructed for each clause Γ in the proof. The interpolant for the proof is then just $I_\bot$. Each clause Γ in the proof is partitioned into $\Gamma_1 \vee \Gamma_2$ with $atoms(\Gamma_2) \subseteq atoms(K_2)$ and $atoms(\Gamma_1) \cap atoms(K_2) = \emptyset$.

The interpolant I_Γ has the property that $K_1 \vdash \neg\Gamma_1 \Rightarrow I_\Gamma$ and $K_2 \vdash I_\Gamma \Rightarrow \Gamma_2$, where $\neg\Gamma_1$ is the set of negations of literals in Γ_1.

For input clauses $\Gamma = \Gamma_1 \vee \Gamma_2$ in K_1, the interpolant $I_\Gamma = \Gamma_2$. For input clauses Γ_2 in K_2, the interpolant is $\top$. When resolving Γ', Γ'' to get Γ,

(1) If resolvent p is in Γ'_1 (i.e., $p \notin atoms(K_2)$), then $I_\Gamma = I_{\Gamma'} \vee I_{\Gamma''}$ since $\neg(p \vee \Gamma'_1) \Rightarrow I_{\Gamma'} \Rightarrow \Gamma'_2$ and $\neg(\neg p \vee \Gamma''_1) \Rightarrow I_{\Gamma''} \Rightarrow \Gamma''_2$.
(2) If resolvent p is in Γ'_2, then $I_\Gamma = I_{\Gamma'} \wedge I_{\Gamma''}$ since $\neg(\Gamma'_1 \vee \Gamma''_2) \Rightarrow I_\Gamma \Rightarrow (p \vee \Gamma'_2) \wedge (\neg p \vee \Gamma''_2) \Rightarrow \Gamma'_2 \vee \Gamma''_2$.

The interpolant for input clause[interpolant] sets $K_1 = \{a \vee e[e], \neg a \vee b[b], \neg a \vee c[c]\}$ and $K_2 = \{\neg b \vee \neg c \vee d[\top], \neg d[\top], \neg e[\top]\}$, with shared atoms b, c, and e can be derived by the following resolution steps.

derive	from
$a[e]$	$a \vee e[e]; e[\top]$
$b[e \vee b]$	$a[e]; \neg a \vee b[b]$
$c[e \vee c]$	$a[e]; \neg a \vee c[c]$
$\neg c \vee d[e \vee b]$	$b[e \vee b]; \neg b \vee \neg c \vee d[\top]$
$d[(e \vee b) \wedge (e \vee c)]$	$\neg c \vee d[e \vee b]; c[e \vee c]$
$\bot[(e \vee b) \wedge (e \vee c)]$	$d[(e \vee b) \wedge (e \vee c)]; \neg d[\top]$

Satisfiability solvers are surveyed by Gomes, Kautz, Sabharwal, and Selman [2008] and in the Handbook of Satisfiability [Biere et al. 2009].

3.3 Satisfiability Modulo Theories

A formula is satisfiable in first-order logic if it has a model. As we already saw in Section 2.2, a theory $\mathcal{T}$ can be defined by a specific class of models so that a sentence is $\mathcal{T}$-satisfiable if it has a model M in $\mathcal{T}$, or it could be defined by its presentation as (a class of models for) a collection of axioms. Theory-based decision procedures have been developed since the late 1970s [Nelson 1981; Shostak et al. 1982], but it is only recently that the techniques used by the DPLL search procedure have been adapted for this purpose.

In SMT, unlike SAT, the atoms are not just Boolean variables but can also represent equalities and applications of various predicates. For example, the set of formulas

$$y = z, \;\; x = y \vee x = z, \;\; x \neq y \vee x \neq z$$

is unsatisfiable due to the interpretation of equality but its propositional skeleton $p, q \vee r, \neg q \vee \neg r$ is satisfiable with the assignment $\{p \mapsto \top, q \mapsto \top, r \mapsto \bot\}$, where p, q, and r represent $y = z$, $x = y$, and $x = z$ respectively. Theory satisfiability can be reduced to SAT by generating lemmas that capture the theory constraints. In the above example, we can add the lemmas $\neg p \vee \neg q \vee r$, $\neg p \vee \neg r \vee q$, and $\neg q \vee \neg r \vee p$. The problem with this *eager* reduction to SAT is that there are 3^n candidate lemmas in n atoms and it is prohibitively expensive to test each of them for theory validity. The *lazy* approach uses SAT to generate a candidate assignment like the one above, which is then refuted by a theory solver.

SMT solvers can deal with other theories including equality over uninterpreted functions, linear arithmetic, bit-vectors, and arrays, as well as combinations of these theories. SMT solvers have been extended to handle quantified formulas through the use of a technique called *E-graph matching* [Nelson 1981]. SMT solvers have a large number of applications since many planning and programming problems can be directly represented as SMT problems. For example, SMT can be used to check the feasibility of a symbolic program path and to generate an actual test case that exercises that path. It can be used to capture quantitative constraints in a planning problem. SMT solvers can be embedded within interactive proof checkers as well as assertion checkers and refinement tools. We only provide a very brief survey of the basic ideas in SMT solving [Nieuwenhuis et al. 2006; Bradley and Manna 2007; de Moura et al. 2007; Kröning and Strichman 2008; Barrett et al. 2009].

We first describe the theory satisfiability procedure TDPLL. Recall that the state of the DPLL procedure is of the form $\langle l, M, K, C\rangle$ with decision level l, partial assignment M, input clause set K, and conflict clause set C. For satisfiability modulo theories, we add a fifth element S which is the theory state. The interaction between the DPLL search and the theory solver is surprisingly simple even if the details of any given implementation can be quite complicated. The interface for the theory solver consists of the *Assert*, *Ask*, *Check*, and *Retract* operations. Whenever a literal is added to M either by selection or propagation, then it is also added to S through the *Assert* operation. We can use *Ask* to check if a particular literal is implied by S, in which case it is added to the partial assignment M. Also, in this case, the theory solver may have the option of generating a *theory lemma* corresponding to this implication which can be added to C. The *Check* operation determines if the state S is inconsistent, in which case the theory solver returns a conflict lemma clause of the form $k_1 \vee \ldots \vee k_n$, where each k_i is the negation of a literal in the *explanation* for the conflict, namely, the set of input literals asserted to S that are relevant to the conflict. In the latter case corresponding to a theory conflict, the lemma clause is added to C and the DPLL procedure also signals a conflict. The conflict is treated as a global conflict if it occurs at level 0, or it triggers backjumping as in the DPLL satisfiability procedure. When literals are dropped from the partial assignment during backjumping, they are also retracted from the theory state S using the *Retract* operation.

An example of the TDPLL search procedure is shown in Figure 9. Here, we consider the previous example of the input clause set $y = z,\ x = y \vee x = z,\ x \neq y \vee x \neq z$. The theory state S here consists of a union-find structure F which maintains the equality information and a set D of the input disequalities. Initially, the partial assignment M and the theory state $\langle F, D\rangle$ are both empty. By DPLL propagation, we add the unit clause $y = z$ to the partial assignment at level 0 and assert it to the theory state. Next, at level 0, we select the literal $x \neq y$ and add it to M and insert it into D. The *scan* step checks all the input literals to collect the ones that are implied or refuted by the theory state. In this case, the literal $x \neq z$ is implied by the theory solver with the supporting lemma $x \neq z \vee y \neq z \vee x = y$. This new literal is added to the partial assignment and the supporting lemma is added to C. Now, DPLL propagation generates a conflict with the clause $x = y \vee x = z$. Analyzing this conflict yields the conflict clause $y \neq z \vee x = y$ which is added to C. Backjumping

Step	M	F	D	C
Prop	$y = z$	$\{y \mapsto z\}$	$\emptyset$	$\emptyset$
Select	$y = z; x \neq y$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\emptyset$
Scan	$\ldots, x \neq z$ $[x \neq z \vee y \neq z \vee x = y]$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\{x \neq z \vee y \neq z \vee x = y\}$
Prop	$\ldots$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\ldots$
Conflict	$\ldots$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\ldots$
Analyze	$\ldots$	$\{y \mapsto z\}$	$\{x \neq y\}$	$\{y \neq z \vee x = y, \ldots\}$
Backjump	$y = z, x = y$	$\{y \mapsto z\}$	$\emptyset$	$\ldots$
Prop	$\ldots, x \neq z[\ldots]$	$\ldots$	$\{x \neq z\}$	$\ldots$
Assert	$y = z, x = y, x \neq z$	$\{x \mapsto y, y \mapsto z\}$	$\{x \neq z\}$	$\ldots$
Check	$y = z, x = y, x \neq z$	$\{x \mapsto y, y \mapsto z\}$	$\{x \neq z\}$	$\ldots$
Conflict				

Fig. 9. Checking the satisfiability of $y = z, \;\; x = y \vee x = z, \;\; x \neq y \vee x \neq z$

with this conflict clause adds the literal $x = y$ at level 0 while retracting the previously asserted literal $x \neq y$. DPLL propagation applied to the input clause $x \neq y \vee x \neq z$ causes $x \neq z$ to be added to M and D. Now, we have a complete assignment and the theory state $\langle F, D\rangle$ is clearly inconsistent.

There are many variations on the basic algorithm described above. For example, the eager approach adds theory lemmas to C prior to the search. The procedure for checking the theory state for inconsistency can be applied at any point during the search or restricted to total assignments. Indeed the checking process can be eliminated in favor of a strong form of *Assert* that detects inconsistencies in the theory state as soon as they are introduced. The theory propagation procedure implemented by querying the theory state for implied literals can be incomplete without affecting the completeness of the search procedure. It can sometimes be more efficient to use a fast but incomplete theory propagation procedure. The operation of scanning the unassigned literals to find an implied literal can either be invoked whenever the theory state is updated or in an intermittent manner. In building an SMT solver, a great deal of experimentation goes into optimizing these parameters to achieve robust performance across the spectrum of benchmarks.

The correctness of the TDPLL inference system is along the same lines as the argument for DPLL. The TDPLL procedure relies on the *Check* procedure being sound and complete, and the *Ask*, *Retract*, and *Assert* procedures being sound. SMT solvers have been around from the late 1970s with the Nelson–Oppen method [Nelson and Oppen 1979; Nelson 1981] used in the Stanford Pascal Verifier and in the Simplify prover [Detlefs et al. 2003] and Shostak's STP [Shostak 1984; Shostak et al. 1982]. More recent SMT solvers are built around the modern versions of the DPLL procedure. In contrast with the lazy approach used in the earlier systems where theory solving is invoked from within a SAT solver, the UCLID [Bryant et al. 2002] system employs an eager combination where theory solving is used to generate theory lemmas that are added to the original formula for checking propositional satisfiability. The lazy combination of theory solving with modern DPLL solvers first appeared in LPSAT [Weld and Wolfman 1999], the Cooperative Validity Checker (CVC) [Stump et al. 2002; Barrett et al. 2002], ICS [de Moura et al. 2002], MathSAT [Audemard et al. 2002], and Verifun [Flanagan et al. 2003]. More recent implementations include Barcelogic [Bofill et al. 2008], CVC3 [Barrett and Tinelli

Delete	$\dfrac{x = y, G; F; D}{G; F; D}$ if $F^*(x) = F^*(y)$
Merge	$\dfrac{x = y, G; F; D}{G; F'; D}$ if $x' = F^*(x) \neq F^*(y) = y'$ $F' = F \cup \{sort(x' = y')\}$
Diseq	$\dfrac{x \neq y, G; F; D}{G; F; x \neq y, D}$
Contrad	$\dfrac{G; F; x \neq y, D}{\bot}$ if $F^*(x) = F^*(y)$

Fig. 10. Inference system for equality and disequality

2007], MathSAT 4 [Bruttomesso et al. 2008], Yices [Dutertre and de Moura 2006b], and Z3 [de Moura and Bjørner 2008].

3.4 Theory Solvers

We have illustrated the TDPLL procedure with a theory solver for equality and disequality based on the union–find algorithm [Galler and Fisher 1964; Tarjan 1975]. This theory solver can itself be seen as an inference system. In this algorithm, the only terms are variables. The inference system is shown in Figure 10. The inference state consists of the input equalities and disequalities G, the *find* map F, and the set of input disequalities D. There is a total ordering $x \succ y$. The map F is represented as a set of equalities $x = y$ such that $x \succ y$, and for any $x = y'$ in F, y is identical to y'. The operation $F(x)$ returns y if there is an equality $x = y$ in F, and x itself, otherwise. The operation $F^*(x) = F^n(x)$ for the smallest n such that $F^n(x) = F^{n+1}(x)$. The operation $sort(x = y)$ returns $x = y$ if $x \succ y$, and $y = x$, otherwise. It is easy to check that inference system for equality is conservative, progressive, and canonizing.

How do we implement the interface operations for *Assert*, *Ask*, *Check*, and *Retract*? The *Assert* procedure adds an equality or a disequality to the state $\langle F, D\rangle$. The *Ask* procedure checks the truth value of an equality $x = y$ against $\langle F, D\rangle$ by checking if $F^*(x) = F^*(y)$ in which case the equality $x = y$ is implied. To determine if $x \neq y$ is implied, we check if there is some disequality $x' \neq y'$ in D such that $F^*(x) = F^*(x')$ and $F^*(y) = F^*(y')$. As an option, we require the *Ask* operation to generate an economical explanation of the implication which can be added as a lemma. Generating such explanations requires a version of union-find where each edge in the find structure corresponds to an input [Nieuwenhuis and Oliveras 2005]. The *Check* operation applies the **Contrad** rule to see if there is an invalid disequality in D. As with *Ask*, generating an explanation for the contradiction requires a proof-carrying version of union-find. Retraction is an easy operation since a disequality can be deleted from D, and for an equality, the corresponding sorted equality can be deleted from F.

Congruence Closure. A theory solver for equality over arbitrary terms can be defined using *congruence closure* [Kozen 1977; Shostak 1978; Nelson and Oppen 1977; Bachmair et al. 2003; Nieuwenhuis and Oliveras 2005]. In this procedure, the *E-graph* data structure used in the union-find algorithm contains nodes for each subterm in the term universe. For each node corresponding to a term the form $f(a_1, \ldots, a_n)$, we have a data structure that maintains the *signature* of the term, namely $f(F^*(a_1), \ldots, F^*(a_n))$. The E-graph is maintained in congruence closed form so that whenever two nodes have the same signature, their equivalence classes are merged. Whenever, there is a disequality $s \neq t$ in D such that $F^*(s) \equiv F^*(t)$, the theory solver signals an inconsistency. The *Assert* command is defined to add an equality or disequality to the E-graph and close it under congruence. The *Ask* command merely checks if an equality $s = t$ is implied by the E-graph or refuted by some disequality in D of the form $s' \neq t'$, where $F^*(s) \equiv F^*(s')$ and $F^*(t) \equiv F^*(t')$. Retraction is as in the union-find inference system.

Linear Arithmetic. There is a wide range of theory solvers for constraint solving with various fragments of linear arithmetic equalities and inequalities. One simple fragment deals only with interval constraints on variables. An inference system for such a fragment can be defined to maintain the tightest interval for each variable. If some variable has an empty interval, then we have a contradiction.

Difference constraints have the form $x - y \leq c$ or $x - y < c$, for some constant c. Strict inequalities $x - y < c$ can be replaced by the non-strict form $x - y \leq c - \epsilon$ by introducing a positive infinitesimal ϵ. Algorithms for processing such constraints include the Bellman–Ford procedure [Wang et al. 2005; Cotton and Maler 2006] and the Floyd–Warshall procedure [Dutertre and de Moura 2006b]. One important twist in the case of SMT solving is that the algorithms must be incremental so that new constraints can be added on-the-fly, and easily retractable so that constraints can be efficiently deleted from the theory solver state in the reverse order in which they were asserted. For difference constraints over integers, the constant c in the constraint must be an integer and we can use 1 instead of the infinitesimal ϵ to replace $x - y < c$ by $x - y \leq c - 1$. The solution, if it exists, must assign integer values to the variables, so that the same algorithms can be applied to both the real and the integer fragment.

In the more general case, we have linear arithmetic constraints of the form $A\vec{x} \leq \vec{b}$, where A is a matrix over the rationals and $\vec{b}$ is a vector of rational constants. Fourier's method [Fourier 1826; Dantzig and Curtis 1973; Williams 1976], also known as the Fourier–Motzkin method, can be used to eliminate a variable, say x_1, from a set of m inequalities of the form $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$, for $1 \leq i \leq m$ by transforming each such inequality into $x_1 \leq (-a_{i2}/a_{i1})x_1 + \ldots + (-a_{in}/a_{i1})x_n + b_i$ for $a_{i1} > 0$, and $x_1 \geq (a_{i2}/ - a_{i1})x_1 + \ldots + (a_{in}/ - a_{i1})x_n + (-b_i)$ for $a_{i1} < 0$. Then for any pair of inequalities of the form $x_1 \leq U$ and $x_1 \geq L$, we add the inequality $L \leq U$ which does not contain x_1. Once all such inequalities have been added, all the inequalities involving x_1 can be deleted. The new inequalities can be transformed to the form $A\vec{x} \leq \vec{b}$ for further variable elimination until we have either a satisfiable or an unsatisfiable set of numeric inequalities. The Omega test [Pugh 1992; Berezin et al. 2003] extends Fourier's method to mixed constraints with both integer and real variables.

Each elimination step of Fourier's method can produces up to $(m/2)^2$ inequalities from m inequalities so that the method is quite expensive in practice. Inference procedures based on the simplex procedure for linear programming have proved effective for the linear arithmetic fragment [Nelson 1981; Detlefs et al. 2003; Vanderbei 2001; Rueß and Shankar 2004]. These procedures demonstrate the infeasibility of $A\vec{x} \leq \vec{b}$ by finding a vector $\vec{y} > 0$ such that $\vec{y}^T A\vec{x} = 1$ and $\vec{y}^T\vec{b} = 0$, where $\vec{y}^T$ is the transpose of $\vec{y}$. Linear arithmetic equalities $s = t$ can be reduced to the equivalent conjunction $s - t \leq 0 \wedge t - s \leq 0$. The general form simplex has been particularly effective for the full linear arithmetic fragment [Dutertre and de Moura 2006a]. Here, input inequalities of the form $s \leq c_u$ or $s \geq c_l$ are converted into tableau entries of the form $x_s = s$ with a freshly chosen variable

x_s corresponding to each canonical polynomial s in the input. We thus have a tableau of the form $\vec{x} = A\vec{y}$, where the variables in $\vec{x}$ are the *basis* variables and the variables in $\vec{y}$ are disjoint from those in $\vec{x}$ and constitute the non-basis variables. The algorithm then maintains the upper and lower bounds $U(x)$ and $L(x)$ for each variable x in the tableau. Whenever $s \leq c_u$ is added and $c_u < U(x_s)$, we update $U(x)$ as c_u. The algorithm also maintains an assignment β for the non-basis variables from which the assignment for the basis variables is computed. Whenever a basis variable has a computed assignment that violates its bounds, pivoting is used to exchange basis and non-basis variables so as to find a new assignment β' that does satisfy the bounds. An inconsistency is determined when there is no suitable pivot candidate for a basis variable whose assignment violates its bounds. The main advantage of this general form algorithm is that retraction has very little cost since we can retain the existing assignment to the variables. The explanation for an inconsistency can be easily constructed from the simplex tableau.

For the case of linear arithmetic constraints with both integer and real variables, the above procedure must be supplemented with heuristic methods since complete procedures can be quite expensive. Strict input inequalities $s < c$ that contain all integer variables can be replaced by $s \leq \lceil c - 1 \rceil$ during pre-processing. For a tableau entry of the form $x = a_1 x_1 + \ldots + a_n x_n$, where the variables all range over the integers and the coefficients are also integers, we can use the *GCD test* to check that the interval for x contains a multiple of the greatest common divisor of the coefficients. For example, if we have $x = 3y - 3z$ with $U(x) = 2$ and $L(x) = 1$, we know that the constraints are not feasible. If the coefficients a_i are non-integer rationals, then the tableau entry can be normalized so that we have $bx = a'_1 x_1 + \ldots + a'_n x_n$, where $a'_i = a_i/b$ and we can replace bx by x' while noting that x' must be divisible by b. The *branch-and-bound* method is invoked whenever an integer variable x has a non-integer assignment $\beta(x)$ to check if the constraints are feasible when conjoined with either of $x \leq \lfloor c \rfloor$ or $x \geq \lceil c \rceil$.

Nonlinear Arithmetic. The Gröbner basis algorithm [Buchberger 1976] can be used to solve the uniform word problem for algebraically closed fields, i.e., fields where every univariate polynomial of non-zero degree has a root. The algorithm works more generally with polynomial rings. Given a set of polynomials Π, the *ideal* generated by Π is the smallest set I with $\Pi \subseteq I$ and $0 \in I$ that is closed under addition and multiplication with any polynomial. Given a set of polynomials and an ordering on the variables, the monomials can be ordered lexicographically so that, for example, if $x \succ y \succ z$, then $x^2yz \succ x^2y \succ xy^2z \succ xyz^2 \succ xy$. This ordering can be lifted lexicographically to polynomials. One polynomial $aM + P$ can be reduced by another polynomial $bN + Q$ for pure (i.e., with coefficient equal to 1) monomials M and N with $N \succ Q$, nonzero coefficients a and b, and polynomials P and Q, if $M = M'N$ for some monomial M'. The reduction replaces the polynomial $aM + P$ by $-aM'Q + bP$. Similarly, the superposition of two polynomials $aM + P$ and $bN + Q$ with $M \succ P$ and $N \succ Q$ adds the polynomial $aM'Q - bN'P$, where M' and N' are the least polynomials such that $MM' = NN'$. Superposition must not be applied to a pair of polynomials where one polynomial can be reduced by the other. The set obtained by starting with Π and repeatedly applying reduction and superposition yields a set of mutually irreducible polynomials B which is the Gröbner basis. A polynomial p is a member of the ideal generated by Π if it is reducible to 0 by the polynomials in B.

To show that $(\bigwedge_{i=1}^{n} p_i = 0) \Rightarrow p = 0$, we start with the set Π of the original set of polynomials augmented with $py - 1$ for some fresh variable y, and apply reduction and superposition to closure while deleting trivial polynomials of the form 0, to obtain the Gröbner basis B of Π. By Hilbert's *Weak Nullstellensatz* [Basu et al. 2003], the ideal contains 1 iff Π has no solution. Therefore, if B contains 1, then Π has no solution. Either $p_1 = 0, \ldots, p_n = 0$ have no solution, and trivially $(\bigwedge_i p_i = 0) \Rightarrow p = 0$, or there is a solution $\overline{X}$. If $p\{\overline{x} \mapsto \overline{X}\} \neq 0$, then there is some assignment Y to y such that $p\{\overline{x} \mapsto \overline{X}, y \mapsto Y\} = 1$, and hence X is also a solution for $py - 1 = 0$, and hence 1 cannot be contained in B. Hence, $(\bigwedge_i p_i = 0) \Rightarrow p = 0$. Conversely, if $\overline{X}, Y$ is a solution for $py - 1$, then clearly $p\{\overline{x} \mapsto \overline{X}\} \neq 0$ and again, 1 is not in the ideal generated by Π, nor in the corresponding basis B.

The Gröbner basis algorithm does not apply to ordered fields like the real numbers. The first-order theory of reals, i.e., the set of first-order sentences true in the reals, is indistinguishable from the theory of *real closed fields*, i.e., ordered fields where non-negative elements have square roots and polynomials of odd degree have roots. Tarski [1948] gave a decision procedure for the first-order theory of real-closed fields which has been improved by Cohen [1969] and Hörmander [1983], and by Collins [1975]. Tiwari [2005] has developed a semi-decision procedure for the universal fragment of real closed fields combining the simplex algorithm and Gröbner basis computations.

Arrays. The clausal validity problem for the extensional theory of arrays given on page 16 can be solved by lazily instantiating the axioms [Dutertre and de Moura 2006b]. For example, whenever $a \neq b$ is asserted, a fresh Skolem constant k is generated along with the lemma $select(a, k) \neq select(b, k) \vee a = b$. Whenever the array term $update(a, i, v)$ appears in the E-graph, we add the lemma $select(update(a, i, v), i) = v$. Additionally, if term b is in the same equivalence class as a or $update(a, i, v)$, then for any term $select(b, j)$ in the E-graph, we add the lemma $i = j \vee select(update(a, i, v), j) = select(a, j)$.

Bit Vectors. The theory of bit-vectors deals with fixed-width bit-vectors and the bit-wise logical operations, various left and right shift operators, as well as signed and unsigned arithmetic operations. If an n-bit bit-vector b is $\langle b_{n-1}, \ldots, b_0\rangle$, then the unsigned interpretation $uval(b)$ is $2^{n-1}b_{n-1} + \ldots + 2^0 b_0$ and the signed (two's complement) interpretation is $uval(\langle b_{n-2}, \ldots, b_0\rangle) - b_{n-1}2^{n-1}$. A simple approach to a bit-vector solver is to *bit-blast* the expression by replacing each bit-vector term b by n bits $\langle b_{n-1}, \ldots, b_0\rangle$ and translating all the operations into the bit representations. This can be expensive and should be done only as needed. Bit-vector problems that require only equality reasoning can be handled efficiently within the E-graph itself.

Combining Theory Solvers. In applying SMT solvers to problems in verification, one typically finds verification problems that span multiple theories including arrays, arithmetic, uninterpreted function symbols, and bit-vectors. The Nelson–Oppen method [Nelson and Oppen 1979; Nelson 1981; Oppen 1980] is a general approach for combining multiple theory solvers as long as the theories involved are over disjoint signatures. A cube ϕ over a combined signature $\Sigma_1 \cup \Sigma_2$, where $\Sigma_1 \cap \Sigma_2 = \emptyset$, is satisfiable in the union of theories if it has a model M whose projection to signature Σ_i is a structure in theory i, for $i = 1, 2$. The first step is to *purify* the formula into an equisatisfiable conjunction $\phi_1 \wedge \phi_2$, where each ϕ_i is a cube entirely in the signature Σ_i. Purification is done in stages by replacing a *pure* subterm s of ϕ in theory i by a fresh variable x while conjoining $x = s$ to the formula. Eventually, we have a cube of the form $\phi' \wedge \bigwedge_{i=1}^{n} x_i = s_i$, where each literal in the cube ϕ' is a pure formula in one of the theories and each equality $x_i = s_i$ is also pure. This conjunction can then be easily partitioned into $\phi_1 \wedge \phi_2$.

We could now apply the individual theory solvers to check if each ϕ_i is satisfiable in theory i, but this does not guarantee satisfiability in the union of the theories since the individual models might not be the projections of a single model. To ensure that the models are *compatible*, we guess an *arrangement* A of the *shared* variables in $vars(\phi_1) \cap vars(\phi_2)$. An arrangement is a conjunction of equalities and disequalities between these variables corresponding to some partition of the variables into equivalence classes so that we have $x = y$ for any two variables x and y in the same equivalence class and $x \neq y$ for any two variables x and y in distinct equivalence classes. We can then check that there is some arrangement A among the finitely many arrangements, such that $\phi_i \wedge A$ is satisfiable in theory i for $i = 1, 2$. If there is such an arrangement, then it guarantees that the original formula ϕ is satisfiable in the union of the theories provided the theories in question are *stably infinite*: if a formula has a model, it has a countably infinite one. Without this proviso, we might still have an incompatibility since the finite cardinalities of the two models might not match. The two countable models M_1 and M_2 that agree on an arrangement can be amalgamated into a single model M for $\phi_1 \wedge \phi_2$ by defining a bijection h between M_2 and M_1 such that $|M| = |M_1|$, $M(f) = M_1(f)$ for $f \in \Sigma_1$, and $M(g)(a_1, \ldots, a_n) = h(M_2(g)(h^{-1}(a_1), \ldots, h^{-1}(a_n)))$ for $g \in \Sigma_2$, and $M(x) = M_1(x) = h(M_2(x))$ for each shared variable x. Conversely, if the formula $\phi_1 \wedge \phi_2$ is satisfiable, then this model yields an arrangement A such that each $\phi_i \wedge A$ is satisfiable in theory i.

For example, the literal

$$select(update(a, 2i + 1, select(a, 3i - 1)), i + 3) \neq select(a, i + 3)$$

can be purified to ϕ_1 of the form $select(update(a, x, select(a, y)), z) \neq select(a, z)$ and ϕ_2 of the form $x = 2i + 1 \wedge y = 3i - 1 \wedge z = i + 3$. The shared variables are x, y, and z, and it can be checked that there is no arrangement A where $\phi_1 \wedge A$ is satisfiable in the theory of arrays and $\phi_2 \wedge A$ is satisfiable in the theory of integer linear arithmetic.

E-graph Matching. While there is no complete method for handling first-order quantification, there are some useful heuristic approaches for instantiating quantifiers in order to derive unsatisfiability within the context of an SMT solver. The E-graph matching method [Nelson 1981; Detlefs et al. 2003] developed in the Stanford Pascal Verifier is one such approach. The E-graph contains vertices corresponding to terms. A quantified formula ϕ can be Skolemized so that it is of the form $\phi_1 \wedge \ldots \wedge \phi_n$, where each ϕ_i is of the form $\forall \overline{x_i}.\kappa_i$, and κ_i is just a clause.

E-graph matching [Nelson 1981] tries to find a ground instance of the clause κ_i that can be added to the set of clauses in SMT procedure. There are usually infinitely many instances so we need to be selective about adding only those instances that are relevant to the search. However, if we only search for syntactic matches, then many useful instantiations could be overlooked. For example, when matching a rule of the form $f(g(x)) = g(f(x))$, the E-graph term universe might not contain terms of the form $f(g(a))$ or $g(f(a))$, but it might contain terms of the form $f(a)$, where a and some other term $g(b)$ are in the same equivalence class. E-graph matching is able to find such a match so that the instance $f(g(b)) = g(f(b))$ is added to the set of clauses in the SMT procedure. E-graph matching can be controlled by identifying *triggers*, which are sets of terms in the clause covering all the free variables that must be matched before the corresponding instance is added. E-graph matching is an incomplete method for quantifier instantiation compared to those in the next section, but it is effective in conjunction with an SMT solver since instantiation is restricted to terms that are represented in the E-graph.

4. PROOF SEARCH IN FIRST-ORDER LOGIC

The early approaches to proof search [Gilmore 1960; Prawitz 1960] in the late 1950s were based on the Herbrand theorem. It was observed by Prawitz [1960], and also by Herbrand [1930] himself, that the Herbrand instantiation could be constructed by equation solving. This is done by picking a bound k on the Herbrand expansion and k sets $\overline{y}_1, \ldots, \overline{y}_k$ of mutually disjoint variables, and converting the formula $\hat{\psi}\{\overline{x} \mapsto \overline{y}_1\} \wedge \ldots \wedge \hat{\psi}\{\overline{x} \mapsto \overline{y}_k\}$ to disjunctive normal form $\Gamma_1 \vee \ldots \vee \Gamma_w$. For the formula to be unsatisfiable, each disjunct Γ_i, $1 \leq i \leq w$ must contain an atom $p(s_1, \ldots, s_u)$ and its negation $\neg p(t_1, \ldots, t_u)$ generating a constraint of the form $p(s_1, \ldots, s_u) = p(t_1, \ldots, t_u)$. The constraints collected from each disjunct must be solved simultaneously over Herbrand terms. Such equations are solved by *unification* which constructs a single substitution for the equations $s_1 = t_1, \ldots, s_w = t_w$ such that $s_i\sigma$ is syntactically identical to $t_i\sigma$ for $1 \leq i \leq w$.

For example, the claim $\forall x.\exists y.p(x) \wedge \neg p(y)$ is unsatisfiable. Here $\hat{\psi}$ is just $p(x) \wedge \neg p(f(x))$, where the Herbrand expansion $p(z) \wedge \neg p(f(z)) \wedge p(f(z)) \wedge p(f(f(z)))$ with $k = 2$ is propositionally unsatisfiable. This expansion could have been obtained by unification from $p(x_1) \wedge \neg p(f(x_1)) \wedge p(x_2) \wedge \neg p(f(x_2))$.

Robinson's resolution method [Robinson 1965] simplified the application of the Herbrand theorem by

(1) Placing φ in clausal form $\forall\kappa_1 \wedge \ldots \wedge \forall\kappa_m$, where the $\forall\kappa_i$ indicates that the free variables in each clause κ_i are universally quantified.

(2) Introducing a resolution inference rule which generates the clause $\forall(\kappa \vee \kappa')\theta$ from $k \vee \kappa$ and $\neg k' \vee \kappa'$, where θ is the *most general unifier* of k and k'. For θ to be a unifier of k and k', the substituted forms $k\theta$ and $k'\theta$ must be syntactically identical. For θ to be the most general unifier of k and k', it must be the case that for any other unifier θ', there is a substitution σ such that $\theta' = \theta \circ \sigma$. In other words, for any substitution $x \mapsto t'$ in θ', either $x \mapsto t$ in θ and $t' = t\sigma$, or there is no substitution for x in θ and $t' = x\sigma$. The clauses $k \vee \kappa$ and $\neg k' \vee \kappa'$ are assumed to have no variables in common. Note that this extends the propositional binary resolution rule to the first-order case by using unification.

(3) Adding a factoring rule to derive $(k \vee \kappa)\theta$ from $k \vee k' \vee \kappa$, where θ is the most general unifier of k and k'. Otherwise, if we resolve $P(x) \vee P(x')$ with $\neg P(y) \vee \neg P(y')$, we get $P(x') \vee \neg P(y')$ and we would never be able to construct a refutation.

Resolution inferences are repeated until the empty clause is generated. The above example $p(x) \wedge \neg p(f(x))$ generates two clauses $p(x)$ and $\neg p(f(y))$ which can be resolved to yield the empty clause. Since validity in first-order logic is undecidable, resolution can only be a semi-decision procedure. When the input clause set is in fact unsatisfiable, the procedure will eventually terminate with the empty clause. However, when the input is satisfiable, the procedure might not terminate. By the Herbrand theorem, we know that if the formula is unsatisfiable, then some Herbrand expansion of the input clause set is propositionally unsatisfiable. The corresponding propositional resolution refutation can be

Right	$\dfrac{x = y \vee L, x = z \vee K, \Gamma}{y = z \vee L \vee K, x = y \vee L, x = z \vee K, \Gamma}$
Left	$\dfrac{x = y \vee L, x \neq z \vee K, \Gamma}{y \neq z \vee L \vee K, x = y \vee L, x \neq z \vee K, \Gamma}$
EqRes	$\dfrac{x \neq x \vee L, \Gamma}{L, x \neq x \vee L, \Gamma}$ L nonempty
Factor	$\dfrac{x = y \vee x = z \vee L, \Gamma}{x = z \vee y \neq z \vee L, x = y \vee x = z \vee L, \Gamma}$
Contrad	$\dfrac{x \neq x, \Gamma}{\bot}$

Fig. 11. Inference system for Superposition

easily simulated by the first-order resolution procedure above as long as the rules are applied *fairly*, i.e., each applicable instance of the resolution or factoring rule is eventually applied.

The basic resolution system described above has many variants and refinements [Bachmair and Ganzinger 2001]. *Subsumption* is an incomplete syntactic check for whether one clause is implied by another, in which case the former clause is deleted for efficiency. The unification algorithm can be enhanced to incorporate theory reasoning [Stickel 1985]. Associative-commutative unification [Baader and Snyder 2001] and higher-order unification [Dowek 2001] are two such examples.

The resolution rule can itself also be enriched to handle equality (using demodulation, paramodulation, and superposition) [Nieuwenhuis and Rubio 2001] and inequality [Stickel 1985; Manna et al. 1991]. Most modern proof search systems use *superposition* [Bachmair et al. 1992; Nieuwenhuis and Rubio 1992], which applies to clauses that contain equality and disequality literals. Non-equality literals of the form p and $\neg p$ can be rewritten to $p = \top$ and $p \neq \top$, respectively. We briefly introduce an inference system for the simplest form of superposition where the atoms are of the form $x = y$. We assume an ordering $\succ$ on variables. Equalities are kept ordered so that if $x = y$, $x \succ y$. Ordering is lifted to literals so that $x = y \succ x' = y'$ (and $x \neq y \succ x' \neq y'$) iff $x \succ x'$ or $x \equiv x'$ and $y \succ y'$, and $x \neq y \succ x = y'$ for any y, y'. Literals of the form $x \neq x$ are deleted from input clauses. Clauses are maintained in decreasing order, and tautologies containing k and $\overline{k}$ or $x = x$ are deleted from the input. For example, given the order $x \succ y \succ z$, the set $\{y = z, x = y \vee x = z, x \neq y \vee x \neq z\}$ contains three ordered clauses.

The superposition inference system for the pure equality fragment is shown in Figure 11. In each inference step, we either derive a contradiction or add a new clause.

The following derivation shows how a contradiction can be derived from the above clause set by applying the inference rules in Figure 11.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{y = z,\ \ x = y \vee x = z,\ \ x \neq y \vee x \neq z}{x = z \vee y \neq z,\ \ldots}\ \textbf{Factor}}{x \neq z \vee y \neq z,\ \ldots}\ \textbf{Left}}{z \neq z \vee y \neq z,\ \ldots}\ \textbf{Left}}{y \neq z,\ \ldots}\ \textbf{EqRes}}{z \neq z,\ \ldots}\ \textbf{Left}}{\bot}\ \textbf{Contrad}$$

The above rules can be extended to the ground case where we have ground terms instead of variables so that x, y, and z are actually terms r, s, and t, and in the **Right** rule, $r = s \vee L$ is resolved with $t[r] = t' \vee K$ to yield the new clause $t[s] = t' \vee K$. The other rules are similarly generalized to ground terms from variables. The calculus can also be lifted to the non-ground case by using unification instead of syntactic equality [Nieuwenhuis and Rubio 2001].

Theorem provers based on resolution include Otter [McCune 1990], E [Schulz 2002], Snark [Stickel et al. 2000], SPASS [Weidenbach et al. 2002], Vampire [Riazanov and Voronkov 2002], and Prover9 [McCune 2007]. The annual CASC competition (CADE ATP System Competition) [Sutcliffe and Suttner 2006] evaluates the performance of theorem proving systems in various categories of first-order logic with and without equality.

5. INTERACTIVE PROOF CHECKERS

In the previous sections, we have seen different ways in which proof and model construction can be automated. Even with such automation, proof construction remains a difficult challenge that calls on human insight and guidance, and a great deal of experimentation. A lot of the time and effort in constructing proofs is devoted to debugging incorrect definitions, conjectures, and putative proofs. Regardless of the level of automation provided by a theorem proving tool, interactivity is needed for delving into the details of an attempted proof. Interactive proof checking has its origins in the work of McCarthy [1962] and de Bruijn's Automath project [de Bruijn 1970; 1980; Nederpelt et al. 1994]. The technology for interactive proof checking was further developed by Bledsoe [Bledsoe and Bruell 1974], Milner [Milner 1972], Weyhrauch [Weyhrauch and Thomas 1974], and Boyer and Moore [Boyer and Moore 1975]. We briefly survey a few of the systems that are actively used in major verification projects including ACL2 [Kaufmann et al. 2000], Coq [Bertot and Castéran 2004], HOL [Gordon and Melham 1993], Isabelle [Paulson 1994], Nuprl [Constable et al. 1986], and PVS [Owre et al. 1995].

5.1 Maude: A Fast Rewrite Engine

Maude [Clavel et al. 1999] and ELAN [Borovanský et al. 2002] are fast and versatile rewrite engines that can be used for building other semantics-based tools. While the rewrite engines are automatic, interaction is used to develop a formal specification consisting of a series of declarations and definitions. Maude is a successor of the OBJ3 system [Goguen et al. 1987] and is based on rewriting logic where different rewriting steps applied to the same term need not be confluent. Maude's rewriting framework is based on membership equational logic which extends first-order conditional equational logic with subsorts and membership assertions of the form $t : s$ for term t and sort s. Each sort is required to be a subsort of a parent *kind*. The signature, equations $l = r$, conditional equations $l = r, \textit{if } b$, and rewrite rules $l \Longrightarrow r$ are given in a module. Maude also allows terms to be treated as equivalent modulo a theory E with respect to rewriting. Thus a term l' will be rewritten by a rewrite rule $l \Longrightarrow r$ if there is a substitution σ such that $E \vdash \sigma(l) = l'$. This allows the language to capture states as terms and state transitions as rewriting steps.

The Maude rewriter employs term-indexing techniques to achieve high speeds of rewriting. By the use of rewriting logic, Maude can be used to define and explore the computational state and operational semantics of a wide range of models of computation. Maude has several interesting applications in metaprogramming [Clavel et al. 1999], program analysis tools [Meseguer and Rosu 2005], symbolic systems biology [Eker et al. 2003], and the analysis of cryptographic protocols [Escobar et al. 2007].

5.2 ACL2: Recursion and Induction

ACL2 [Kaufmann and Moore 1996; Kaufmann et al. 2000] is the most recent in a line of inductive theorem provers initiated by Boyer and Moore [1979; 1988] in 1973. These provers are built on a computational logic formalizing pure Lisp given by McCarthy [1963]. The ACL2 logic is based on an applicative fragment of the widely used programming language Common Lisp [Steele Jr. 1990]. The theorem prover is itself written in this fragment. It can be used interactively to construct a formal development consisting of datatypes, axioms, definitions, and lemmas. The definitions can be compiled and executed as Common Lisp functions. When a recursive definition is presented to ACL2, the prover attempts to establish the termination of the recursion scheme. It does this by constructing a measure or a size function on the arguments that decreases with each recursive call according to a well-founded ordering, i.e., an ordering without any infinite descending chains. The prover retains an induction scheme, based on the termination ordering, for use in induction proofs, and also infers a small amount of useful type information about the definition for future use.

The ACL2 interactive prover can be used to attempt to prove conjectures. When given a conjecture, the prover tries to prove the theorem using a cascade of simplifications, a *recursive waterfall*, involving equality and propositional reasoning, case analysis, rewriting with definitions and lemmas. Each step can generate zero or more subgoals, each of which is processed from the top of the waterfall. When there are no remaining subgoals, the proof has succeeded. When a conjecture does not succumb to simplification, the prover attempts a proof by induction. The termination arguments for the recursive definitions that appear in the conjecture are used to construct an induction scheme. The sub-cases of the induction proof are subject to the recursive waterfall. Any unproven subgoals are generalized so that a nested induction can be applied to them.

We give a brief example of ACL2 at work. The theories of numbers and lists is built into ACL2. Lists are defined by the constructors `NIL` and `CONS`, where the latter constructor has accessors `CAR` and `CDR`, and a corresponding recognizer `CONSP`. The operation `(ENDP X)` is defined as `(NOT (CONSP X))`. Boolean reasoning is internalized so that the truth value $\top$ is represented by the symbol `T` and $\bot$ is represented by the symbol `NIL`. A formula ϕ with free variables $x_1, \ldots, x_n$ is valid if no ground instance of it is equal to `NIL`.

The recursive definition of `TRUE-LISTP` describes a predicate that holds only when the argument `X` is a list that is terminated by `NIL`. The Common Lisp equality predicate

`EQ` is used to compare `X` to the symbol `NIL`.

```
(DEFUN TRUE-LISTP (X)
       (IF (CONSP X)
           (TRUE-LISTP (CDR X))
           (EQ X NIL)))
```

The operation of reversing a list can be defined as below. It can be shown to terminate by the well-founded ordering on the size of `X`. In general, termination measures can be given using a representation of the ordinals below ϵ_0, the least ordinal α such that $\alpha = \omega^\alpha$. These ordinals and their corresponding ordering can be defined in primitive recursive arithmetic and can describe nontrivial nested lexicographic orderings.

```
(DEFUN REV (X)
       (IF (ENDP X)
           NIL
           (APPEND (REV (CDR X)) (LIST (CAR X)))))
```

The definition can be evaluated at the prompt so that we can check that `(REV '(3 4 5))` evaluates to the list `(5 4 3)`, and `(REV (REV '(3 4 5)))` is `(3 4 5)`. If we now try to prove the conjecture `REV_OF_REV` below, the proof attempt fails.

```
(DEFTHM REV_OF_REV
        (EQUAL (REV (REV X)) X))
```

Since `REV` is a recursive function, ACL2 attempts a proof by induction which eventually yields a subgoal of the form `(IMPLIES (NOT (CONSP X)) (NOT X))` which fails because it is possible for `X` to be a non-`CONSP` without being equal to `NIL`. This claim obviously only holds if `X` is constrained to satisfying the `TRUE-LISTP` predicate. If we now fix the statement of the conjecture, ACL2 is able to prove this automatically.

```
(DEFTHM REV_OF_REV
        (IMPLIES (TRUE-LISTP X)
                 (EQUAL (REV (REV X)) X)))
```

In attempting the proof by induction, ACL2 is able to conjecture that the main induction subgoal requires the lemma below, which it is able to prove directly by induction.

```
(EQUAL (REV (APPEND RV (LIST X1)))
       (CONS X1 (REV RV)))
```

Large sequences of definitions and theorems can be developed in this manner and packaged into files containing definitions and theorems, called *books*. ACL2 has been used in an impressive list of verifications, including several involving commercial systems. These include the verification of floating-point hardware at AMD [Russinoff 1999] and various processor and systems verification efforts at Rockwell-Collins [Greve et al. 2003]. Bundy [2001] describes the various approaches to automating inductive proofs.

5.3 The LCF Family of Tactical Proof Systems

LCF is actually an acronym for Scott's Logic for Computable Functions [Scott 1993] but the name is more closely associated with a class of extensible proof checkers pioneered by Milner [1979]. The programming language ML [Gordon et al. 1977] was developed to serve as the metalanguage for defining proof checkers in the LCF style. The key idea is to introduce a datatype `thm` of theorems. The constructors of this datatype are the inference rules that map `thm list` to `thm`. *Tactics* written in ML are used to convert goals to subgoals so that $\tau(G) = \{S_1, \ldots, S_n\}$ with a validation v such that $v(S_1, \ldots, S_n) = G$. A

```
module Proven : Birkhoff =
struct
  type thm = formula list * formula
  let axiom p =
    match p with
      Atom("=",[s;t]) -> ([p],p)
    | _ -> failwith "axiom: not an equation"
  let inst i (asm,p) = (asm,formsubst i p)
  let refl t = ([],Atom("=",[t;t]))
  let sym (asm,Atom("=",[s;t])) =
        (asm,Atom("=",[t;s]))
  let trans (asm1,Atom("=",[s;t]))
            (asm2,Atom("=",[t';u])) =
    if t' = t then (union asm1 asm2,Atom("=",[s;u]))
    else failwith "trans: theorems don't match up"
  let cong f ths =
    let asms,eqs =
      unzip(map (fun (asm,Atom("=",[s;t]))
           -> asm,(s,t)) ths) in
    let ls,rs = unzip eqs in
    (unions asms,Atom("=",[Fn(f,ls);Fn(f,rs)]))
  let dest_thm th = th
end;;
```

Fig. 12. An LCF-style proof system for equational logic

proof can be constructed *backwards* by applying tactics to goals and subgoals or *forwards* from axioms by building validations using inference rules.

John Harrison [Harrison 2001] presents a simple LCF-style implementation of a proof system for equational logic displayed in Figure 4. First, the thm datatype is introduced with the signature Birkhoff.

```
module type Birkhoff =
sig type thm
val axiom : formula -> thm
val inst : (string, term) func -> thm -> thm
val refl : term -> thm
val sym : thm -> thm
val trans : thm -> thm -> thm
val cong : string -> thm list -> thm
val dest_thm : thm -> formula list * formula
end;;
```

A structure of this signature can then be defined to implement the datatype thm as a subset of sequents of the form $\Gamma \vdash \phi$ that are constructed using the inference rules implemented in Figure 12. The constructor Fn applies a function symbol to a list of arguments, and the constructor Atom applies the equality symbol to a list of two arguments. A sequent $\Gamma \vdash \phi$ is represented as a pair consisting of the list of assumptions Γ and the equality ϕ. As an example of an inference rule, the congruence rule cong takes a list of theorems of the form $\Gamma_1 \vdash s_1 = t_1, \ldots, \Gamma_n \vdash s_n = t_n$ and returns the sequent $\bigcup_i \Gamma_i \vdash f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$.

By composing the constructors for the thm datatype, we can construct derived inference rules in terms of functions that take a list of elements of type thm to a thm. It is also possible to build proofs *backwards* from goal sequents to subgoal sequents through the application of a tactic to the goal as described above. *Tactics* can be defined in the metalanguage ML. The application of a tactic to a goal can generate zero or more subgoals, or terminate with an exception. The application of a tactic could generate an inappropriate validation in which case the proof would eventually fail. Tactics can be composed using *tacticals* such as those for sequencing, alternation, and repetition. Tacticals can also be defined in the metalanguage ML. The LCF approach thus facilitates the construction of interactive proof checkers that can be extended with derived inference rules while preserving soundness relative to a small kernel.

Many proof checkers are based on the LCF approach, including HOL [Gordon and Melham 1993], HOL Light [Harrison 1996], Coq [Bertot and Castéran 2004], Isabelle [Paulson 1994], LEGO [Luo and Pollack 1992], and Nuprl [Constable et al. 1986]. We examine some of these systems in greater detail below.

5.3.1 *HOL and its Variants.* Automated reasoning in higher-order logic was actively investigated by Andrews in his TPS system [Andrews et al. 1988]. The use of higher-order logic in interactive proof checking was initiated by Hanna [1986] and Gordon [1985b; 1985a] to address hardware description and verification. Currently, the most popular higher-order logic proof checkers are HOL4 [Slind and Norrish 2008], HOL Light [Harrison 1996], and Isabelle/HOL [Nipkow et al. 2002]. These proof checkers are widely used in formalizing mathematics and program semantics, and in verifying hardware, distributed algorithms, cryptographic protocols, and floating point algorithms. Recently, Hales [2002; 2009] has initiated the Flyspeck project to verify his computer-based proof of Kepler's conjecture about the optimality of the cannonball packing of congruent spheres. The verification of the proof involves Coq, HOL Light, and Isabelle/HOL.

The HOL Light system, for example, is based on a very small kernel with types for Booleans and individuals, inference rules for equality (reflexivity, symmetry, transitivity, congruence, β-reduction, extensionality, and equivalence), and axioms for infinity and choice. In addition, there is a definition principle for defining new constants. New types can be introduced axiomatically provided they are shown to be interpretable in the existing type system. The higher-order logic admits parametric polymorphism through the use of type variables. The resulting kernel runs to about 500 lines of OCaml code. HOL Light has been used extensively for the verification of floating point hardware algorithms [Harrison 2006] and for formalizing several interesting proofs from various branches of mathematics [Hales 2007] (see also `http://www.cs.ru.nl/~freek/100/hol.html`).

5.3.2 *Nuprl and Coq: Proofs in Constructive Type Theories.* Both Nuprl [Constable et al. 1986] and Coq [Coquand and Huet 1988; Bertot and Castéran 2004] are based on the Curry-Howard isomorphism [Howard 1980] between propositions and types. The implicational fragment of intuitionistic logic offers a simple illustration of this isomorphism. The natural deduction sequent $\Gamma \vdash \phi$ represented by the judgment $x_1 : \gamma_1, \ldots, x_n : \gamma_n \vdash t : \phi$ and asserts that t is a proof term for ϕ from the hypothetical proof terms (variables) $x_1, \ldots, x_n$ corresponding to the assumptions $\Gamma = \gamma_1, \ldots, \gamma_n$. Here, the *context* $x_1 : \gamma_1, \ldots, x_n : \gamma_n$ contains exactly one declaration $x_i : \gamma_i$ for each variable x_i. The introduction rule for implication builds a proof term using lambda-abstraction, whereas the elimination rule uses function application corresponding to the use of the *modus ponens* rule of inference. Thus, complete proofs are represented by well-typed closed terms of the typed lambda calculus, and the propositions proved by these proofs correspond to the types for these terms. A proposition is provable if the corresponding type is *inhabited* by a term.

If the above proof rules are viewed as the type rules of a simply typed lambda calculus, then the formula $\phi_1 \Rightarrow \phi_2$ corresponds to a type $A_1 \rightarrow A_2$. The simply typed lambda

$\dfrac{}{x_1 : \phi_1, \ldots, x_n : \phi_n \vdash x_i : \phi_i}$	$\dfrac{\Gamma \vdash u : \phi_1 \qquad \Gamma \vdash t : \phi_1 \Rightarrow \phi_2}{\Gamma \vdash tu : \phi_2}$	$\dfrac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda(x : \phi_1).t : \phi_1 \Rightarrow \phi_2}$

Fig. 13. Proof terms for natural deduction

$\dfrac{\Gamma \vdash u : \phi_1 \wedge \phi_2}{\Gamma \vdash \mathit{fst}(u) : \phi_1}$	$\dfrac{\Gamma \vdash u : \phi_1 \wedge \phi_2}{\Gamma \vdash \mathit{snd}(u) : \phi_2}$	$\dfrac{\Gamma_1 \vdash u_1 : \phi_1 \qquad \Gamma_2 \vdash u_2 : \phi_2}{\Gamma_1, \Gamma_2 \vdash (u, v) : \phi_1 \wedge \phi_2}$

Fig. 14. Proof terms for conjunction

$\dfrac{\Gamma \vdash u : \phi_1 \qquad \Gamma \vdash t : (\forall(x : \phi_1).\phi_2)}{\Gamma \vdash tu : \phi_2\{x \mapsto u\}}$	$\dfrac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash (\lambda(x : \phi_1).t) : (\forall(x : \phi_1).\phi_2)}$

$\dfrac{\Gamma \vdash u : (\exists x : \phi_1.\phi_2)}{\Gamma \vdash \mathit{fst}(u) : \phi_1}$	$\dfrac{\Gamma \vdash u : (\exists x : \phi_1.\phi_2)}{\Gamma \vdash \mathit{snd}(u) : \phi_2\{x \mapsto \mathit{fst}(u)\}}$	$\dfrac{\Gamma \vdash t : \phi_1 \qquad \Gamma \vdash u : \phi_2\{x \mapsto t\}}{\Gamma \vdash (t, u) : \exists x : \phi_1.\phi_2}$

Fig. 15. Proof terms for quantification

calculus can be extended in several directions. One extension is to add conjunction $\phi_1 \wedge \phi_2$ with the introduction and elimination rules shown in Figure 14. The conjunction $\phi_1 \wedge \phi_2$ then corresponds to the product type $A \times B$. Similarly, intuitionistic disjunction can be presented with rules that correspond to the typing rules for the disjoint union $A + B$ over types A and B.

One can extend the proof/type rules in the direction of first-order logic by introducing *dependent types*. Dependent product types are represented as $(\forall(x : A).B(x))$ and characterize those functions that map elements a of type A to elements of type $B(a)$. Dependent sum types are represented as $(\exists(x : A).B(x))$, and capture pairs of elements (a, b) such that a is an element of type A and b is an element of type $B(a)$. The type rules for implication and conjunction can be modified as shown in Figure 15. In particular, the function type $A{\rightarrow}B$ is just $(\forall(x : A).B)$, where B does not contain a free occurrence of x, and similarly $A \times B$ is just $(\exists(x : A).B)$, where B does not contain a free occurrence of x.

The typed lambda calculus can be extended along a different dimension to allow type abstraction. Dependent typing allows types to be parameterized by terms, whereas polymorphism allows both types and terms parameterized by types. Let $*$ represent the *kind* of types so that $\Gamma \vdash (\forall(x : A).B) : *$ follows from $\Gamma \vdash A : *$ and $\Gamma, x : A \vdash B : *$. Then the type of the polymorphic identity function $(\lambda(\alpha : *).(\lambda(x : \alpha).x))$ is $(\forall(\alpha : *).\alpha{\rightarrow}\alpha)$. Polymorphism can also be used introduce representations for inductive types like natural numbers and lists. We can also define the other logical connectives using type quantification.

$$\begin{aligned}
A{\rightarrow}B &= (\forall(x : A).B), x \text{ not free in } B \\
A + B &= (\forall(C : *).(A{\rightarrow}C){\rightarrow}(B{\rightarrow}C){\rightarrow}C) \\
(\exists(A : *).B) &= (\forall(C : *).(\forall(x : A).B{\rightarrow}C){\rightarrow}C)
\end{aligned}$$

There is one further dimension along which the expressiveness of the calculus can be increased. While the introduction of the kind $*$ yielded types of the form $\forall(A : *).A{\rightarrow}A$, we still do not have lambda-abstraction with respect to type variables. For example, we cannot construct $\lambda(A : *).A{\rightarrow}A$. For this purpose, the notation $\Box$ is introduced to rep-

$\dfrac{}{\vdash * : \Box}$	
$\dfrac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A}$	$\dfrac{\Gamma \vdash \kappa : \Box}{\Gamma, x : \kappa \vdash x : \kappa}$
$\dfrac{\Gamma, x : A \vdash B : *}{\Gamma \vdash (\forall(x : A).B) : *}$	$\dfrac{\Gamma, x : A \vdash \kappa : \Box}{\Gamma \vdash (\forall(x : A).\kappa) : \Box}$
$\dfrac{\Gamma, x : A \vdash B : * \qquad \Gamma, x : A \vdash s : B}{\Gamma \vdash (\lambda(x : A).s) : (\forall(x : A).B)}$	$\dfrac{\Gamma, x : A \vdash \kappa : \Box \qquad \Gamma, x : A \vdash B : \kappa}{\Gamma \vdash (\lambda(x : A).B) : (\forall(x : A).\kappa)}$
$\dfrac{\Gamma \vdash s : (\forall(x : A).B) \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B\{x \mapsto t\}}$	$\dfrac{\Gamma \vdash s : (\forall(x : A).\kappa) \qquad \Gamma \vdash t : A}{\Gamma \vdash st : \kappa\{x \mapsto t\}}$
$\dfrac{\Gamma \vdash s : A \qquad \Gamma \vdash B : * \qquad A =_\beta B}{\Gamma \vdash s : B}$	$\dfrac{\Gamma \vdash A : \kappa \qquad \Gamma \vdash \kappa' : \Box \qquad \kappa =_\beta \kappa'}{\Gamma \vdash A : \kappa'}$

Fig. 16. The Calculus of Constructions

resent the class of well-formed kinds so that the typing judgment $\lambda(A : *).A {\to} A : * {\to} *$ holds, where $* {\to} * : \Box$. These three dimensions of term-parametric types (dependent typing), type-parametric terms (polymorphism), and type-parametric types (type constructors) form the three dimensions of Barendregt's cube [1992] of typed lambda-calculi that satisfy *strong normalization*: all reduction sequences terminate. The Calculus of Constructions [Coquand and Huet 1988] is the most expressive of these calculi. It is based on a typed lambda calculus that integrates both dependent typing and polymorphism, while admitting type abstraction in constructing types. In this calculus, the polymorphic dependently typed lambda calculus is augmented rules for $\Box$ such that $\vdash * : \Box$ holds and $\Gamma \vdash (\forall(x : \alpha).\beta) : \Box$ if $\Gamma \vdash \alpha : \Box$ and $\Gamma, x : \alpha \vdash \beta : \Box$. If two types A and B, when fully β-reduced, are identical modulo the renaming of bound variables, then any term of type A also has type B. We then obtain a type system shown in Figure 16 that is the foundation for the Coq proof checker [The Coq Development Team 2009]. The system also includes a mechanism for directly defining inductive types [Paulin-Mohring 1993].

The Coq system has been used in several significant proof checking exercises including a complete proof of the four color theorem [Gonthier 2008], Gödel's first incompleteness theorem [O'Connor 2005], the correctness of a compiler for a subset of the C language [Leroy 2007], and the correctness of various inference procedures [Théry 1998].

The logic of the Nuprl proof checker is based on Martin-Löf's intuitionistic type theory [Martin-Löf 1980]. Nuprl employs Curry's version of the typed lambda calculus where the variables in the terms are not restricted to specific type, but type inference is used to derive the types. Propositions in this type system are built from basic types such as `int` and `void` (the empty type) using *dependent products* $(\forall(x \in A).B(x)$ corresponding to universal quantification, and *dependent sums* $(\exists(x \in A).B(x)$ corresponding to existential quantification. Other type constructors include the subset type $\{x : A \mid B\}$ which contains elements a of type A such that the type $B(a)$ is inhabited, and the quotient type $A//E$, where E is an equivalence relation on A. The type `atom` of character strings and the type A `list` of lists over A are also included in the type system. For any type A and terms a and b of type A, the expression $a = b \in A$ is also a type. There is also a primitive type $a < b$ for a and b of type `int`. Finally, there is a cumulative hierarchy of type universes $U_1, \ldots, U_i, \ldots$, where each U_i is a term of type U_{i+1} and every term of type U_i is also of type U_{i+1} for $i > 0$. Nuprl has been used in optimizing the protocol stack for a high-performance group communication system [Liu et al. 1999].

5.4 Logical Frameworks: Isabelle, λ-Prolog, and Twelf

A logical framework is a way of defining a wide range of object logics while sharing the implementation of basic operations like substitution. When representing the syntax of object logics, there is a choice between first-order abstract syntax using Lisp-style s-expressions or higher-order abstract syntax [Harper et al. 1987; Pfenning 2001] that employs lambda abstraction at the metalogical level as a way of capturing substitution at the object level. Logical frameworks such as λ-Prolog [Miller and Nadathur 1986; Nadathur and Miller 1990], Isabelle [Paulson 1994], and Twelf [Pfenning and Schürmann 1999] employ higher-order abstract syntax. Logics are represented in a small subset of higher-order intuitionistic logic similar to the Horn fragment used in logic programming. This way, if Φ is the type of formulas in the object language, then $\wedge$, $\vee$, and $\Rightarrow$ in the object logic can be represented by constructors `and`, `or`, and `implies` with the type $\Phi{\rightarrow}\Phi{\rightarrow}\Phi$, and $\neg$ by `not` with the type $\Phi{\rightarrow}\Phi$. Interestingly, however, universal and existential quantification can be represented by higher-order constructors `forall` and `exists` of type $i{\rightarrow}\Phi$, where i is the type of individuals in the higher-order metalogic.

The next step is to represent proofs. One approach is to represent a proof predicate $\texttt{proof}(\pi, \phi)$ to represent the assertion that π is a proof of ϕ. Then, we can represent the proof rules as logic programs so that the natural deduction rules for implication are as follows.

$$\begin{array}{l}
\texttt{proof}(\texttt{imp\_i}(P1), \texttt{implies}(A, B)) \\
\quad \texttt{:-}\ (\forall P. \texttt{proof}(P, A) \Rightarrow \texttt{proof}(P1(P), B)). \\
\texttt{proof}(\texttt{imp\_e}(P1, P2, A), B) \\
\quad \texttt{:-}\ \texttt{proof}(P1, \texttt{implies}(A, B)), \texttt{proof}(P2, A).
\end{array}$$

Note that the antecedent of the introduction rule uses universal quantification and implication. The logic programming fragment used here is therefore more expressive than the Horn clause fragment used by Prolog. The introduction and elimination rules for universal quantification can also be transcribed as logic programming clauses.

$$\begin{array}{rl}
\texttt{proof}(\texttt{forall\_i}(P), \texttt{forall}(A)) & \texttt{:-}\ (\forall (c : i). \texttt{proof}(P(c), A(c))). \\
\texttt{proof}(\texttt{forall\_e}(P, t), A(t)) & \texttt{:-}\ \texttt{proof}(P, \texttt{forall}(A)).
\end{array}$$

The Isabelle logical framework [Paulson 1994] uses intuitionistic higher-order logic with implication, universal quantification, and equality with a resolution strategy for constructing object-level proofs by means of theorem proving at the meta-level. Many different object logics have been formalized in Isabelle, but ZF set theory and higher-order logic (HOL) are the ones that are most developed. Isabelle has a number of generic interfaces for defining simplifiers and other inference tools, including a tableau-based proof search engine for first-order logic. Isar [Wenzel 1999] is a declarative style of proof presentation and verification for Isabelle inspired by the Mizar proof checking system [Rudnicki 1992]. Isabelle/HOL has been used for verifying a number of cryptographic protocols [Paulson 1998]. It is also used within the Flyspeck project [Hales 2002; Hales et al. 2009], the Verisoft project for the pervasive verification of a distributed real-time system [Knapp and Paul 2007], and the ongoing verification of the seL4 microkernel [Elkaduwe et al. 2008].

The λ-Prolog [Nadathur and Miller 1990] logical framework uses hereditary Harrop formulas to define a logic programming engine. The Teyjus logic programming framework [Nadathur and Mitchell 1999] implements this form of higher-order logic programming. The Twelf logical framework [Pfenning and Schürmann 1999] employs dependent typing and a *propositions as judgments* interpretation of intuitionistic logic.

5.5 PVS: Integrating Type Systems and Decision Procedures

The Prototype Verification System (PVS) [Owre et al. 1995] occupies the middle ground between a highly automated theorem prover like ACL2 and an interactive checker for formal proofs in the LCF style. In particular, PVS exploits the synergy between an expressive specification language and an interactive proof checker that can be used to develop proof scripts that integrate several automated tools. The PVS specification language is based on higher-order logic enhanced with predicate subtypes (similar to the subset type from Nuprl), dependent types, structural subtypes (where a record can have more fields than its supertype), inductive and coinductive datatypes, parametric theories, and theory interpretations, and The PVS proof checker builds on a range of decision procedures including SAT and SMT procedures [Shostak et al. 1982; Dutertre and de Moura 2006b], binary decision diagrams, symbolic model checking [Rajan et al. 1995], predicate abstraction [Saïdi and Graf 1997; Saïdi and Shankar 1999], and decision procedures for monadic second-order logic and Presburger arithmetic (MONA) [Elgaard et al. 1998]. Even with automation and scripting, interaction is still needed in PVS to direct the induction, case analysis, definition expansion, and quantifier instantiation steps needed to build readable proofs.

A significant subset of the PVS language is executable as a functional language. The code generated from this subset includes optimizations for in-place updates. The PVS framework is open so that new inference tools can be plugged into the system. PVS has been used as a back-end inference framework in a number of tools including TLV [Pnueli and Shahar 1996], Why [Filliâtre and Marché 2007], LOOP [van den Berg and Jacobs 2001], Bandera [Corbett et al. 2000], PVS-Maple [Adams et al. 2001], TAME [Archer and Heitmeyer 1996], and InVest [Bensalem et al. 1998]. PVS has also been applied in a number of significant verification exercises covering distributed algorithms [Miner et al. 2004], hardware verification [Rueß et al. 1996], air-traffic control [Carreño and Muñoz 2000], and computer security [Millen and Rueß 2000].

6. LOOKING AHEAD

Logic is a fertile semantic foundation for writing specifications, building models, and capturing program semantics. It has been used this way in verification for a very long time, but recent advances in the level of automation have made it possible to apply these techniques in a scalable way to realistic software. These automated tools include satisfiability procedures, rewriting engines, proof search procedures, and interactive proof checkers. Individual tools will of course continue to gain in power and expressiveness. They will also find novel applications in verification as well as in areas like artificial intelligence and computer-aided design. The major advances will be in the integration of heterogeneous deductive capabilities, including

(1) Semantic interoperability between different inference procedures and logics through a semantic tool bus. Many interesting analyses require cooperation between satisfiability procedures, static analysis tools, model checkers, rewriters, and proof search

engines for the purpose of generating and checking abstractions, assertions, and termination ranking functions.

(2) *Better integration of constraint solving, matching, and unification* [Baader and Snyder 2001]. There has already been a lot of work in adding associative-commutative operations and higher-order unification [Dowek 2001], but there is a rich set of theories such as arithmetic, encryption, arrays, and partial orders, where unification enhanced with constraint solving can be very effective.

(3) *Satisfiability under quantification.* This is another area where there is plenty of scope for dramatic improvements. Quantified Boolean Formulas (QBF) augment propositional formulas with Boolean quantification. QBF satisfiability procedures are being actively developed for a variety of applications. Proof search procedures for quantified formulas that exploit SMT solvers can be significantly more effective than pure first-order proof search [Stickel 1985; Ganzinger and Korovin 2006].

(4) *Integrating deduction, abstraction, static analysis, and model checking*. The companion survey by Jhala and Majumdar [2009] covers the key ideas in software model checking, analysis, and verification. Recent systems like Spec# [Barnett et al. 2005], BLAST [Henzinger et al. 2003], TVLA [Bogudlov et al. 2007], Bohne [Wies et al. 2006], and Dash/Yogi [Beckman et al. 2008] incorporate diverse tools for analysis and deduction.

(5) *Experimental evaluation and benchmarking.* Competitions such as CASC [Sutcliffe and Suttner 2006], SAT [Simon et al. 2005], and SMT [Barrett et al. 2005] competitions have led to the accumulation of a useful body of benchmarks for optimizing inference procedures. A lot of the recent progress in inference procedures can be attributed to these competitions and the standardized benchmarks.

(6) *Fine-grained classification of feasibility.* Many useful fragments of logic have satisfiability problems that are either infeasible or undecidable, yet there are many instances where this does not pose an insurmountable obstacle to practical use. There is clearly a lot of work that needs to be done in characterizing the class of problems that are feasibly solvable along with techniques that are effective in practice.

(7) *Synthesis.* The synthesis of formulas satisfying certain constraints including interpolants, invariants, abstractions, interfaces, protocols, ranking functions, winning strategies, and ruler-and-compass constructions.

7. CONCLUSIONS

Software verification is a challenging problem because it relates three complex concepts: software, properties, and proofs. Automated deduction is an important tool for stating and verifying properties of software, supporting stepwise program refinement, generating test cases, and building evidence supporting formal claims for software correctness. Recent years have seen exciting progress in the automation and efficiency of theorem provers as well as in novel applications of automated deduction techniques. While automated deduction is a highly developed discipline with a sophisticated range of tools and techniques, there is a coherence to the core ideas that we have presented here. This is illustrated, for example, in the way that the DPLL satisfiability procedure generates proofs based on resolution. Automated deduction could fruitfully interact with other disciplines like philosophy, biology, economics, knowledge representation, databases, programming languages, and linguistics. Vannevar Bush, in his prescient 1945 article *As We May Think*, predicted that

> *Logic can become enormously difficult, and it would undoubtedly be well to produce more assurance in its use. . . . We may some day click off arguments on a machine with the same assurance that we now enter sales on a cash register.*

This prediction, like his other ones, may yet prove accurate, but for software verification, automated deduction is already a critical technology.

Acknowledgments. Tony Hoare and Jayadev Misra suggested the idea for this survey article and shepherded it through many revisions with copious feedback, advice, and encouragement. Bruno Dutertre, Jean-Christophe Filliâtre, John Harrison, David Naumann, Sam Owre, John Rushby, Andreas Podelski, Alexandre Rademaker, Mark-Oliver Stehr, and Ashish Tiwari carefully read through earlier drafts and suggested many improvements. Bill McCune and Matt Kaufmann responded with clarifications on specific points. The article has been significantly improved by the detailed, authoritative, and deeply knowledgeable comments from the anonymous referees.

REFERENCES

ABDULLA, P. A., ČERĀNS, K., JONSSON, B., AND TSAY, Y.-K. 1996. General decidability theorems for infinite-state systems. In *Proceedings, 11*$^{\text{th}}$ *Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, New Brunswick, New Jersey, 313–321.

ABRAMSKY, S., GABBAY, D. M., AND MAIBAUM, T. S. E., Eds. 1992a. *Handbook of Logic in Computer Science; Volume 1 Background: Mathematical Structures*. Oxford Science Publications, Oxford, UK.

ABRAMSKY, S., GABBAY, D. M., AND MAIBAUM, T. S. E., Eds. 1992b. *Handbook of Logic in Computer Science; Volume 2 Background: Computational Structures*. Oxford Science Publications, Oxford, UK.

ABRIAL, J.-R. 1980. *The Specification Language Z: Syntax and Semantics*. Programming Research Group, Oxford University, Oxford, UK.

ABRIAL, J.-R. 1996. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

ADAMS, A., DUNSTAN, M., GOTTLIEBSEN, H., KELSEY, T., MARTIN, U., AND OWRE, S. 2001. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In *Theorem Proving in Higher Order Logics, TPHOLs 2001*, R. J. Boulton and P. B. Jackson, Eds. Lecture Notes in Computer Science, vol. 2152. Springer-Verlag, Edinburgh, Scotland, 27–42.

AMLA, N., DU, X., KUEHLMANN, A., KURSHAN, R. P., AND MCMILLAN, K. L. 2005. An analysis of SAT-based model checking techniques in an industrial environment. In *Correct Hardware Design and Verification Methods: 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005*, D. Borrione and W. Paul, Eds. Lecture Notes in Computer Science, vol. 3725. Springer-Verlag, Saarbrücken, Germany, 254–268.

ANDREWS, P. B. 1986. *An Introduction to Logic and Type Theory: To Truth through Proof.* Academic Press, New York, NY.

ANDREWS, P. B., ISSAR, S., NESMITH, D., AND PFENNING, F. 1988. The TPS theorem proving system. In *9th International Conference on Automated Deduction (CADE)*, E. Lusk and R. Overbeek, Eds. Lecture Notes in Computer Science, vol. 310. Springer-Verlag, Argonne, IL, 760–761.

ARCHER, M. AND HEITMEYER, C. 1996. TAME: A specialized specification and verification system for timed automata. In *Work In Progress (WIP) Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, A. Bestavros, Ed. Washington, DC, 3–6. The WIP Proceedings is available at `http://www.cs.bu.edu/pub/ieee-rts/rtss96/wip/proceedings`.

AUDEMARD, G., BERTOLI, P., CIMATTI, A., KORNILOWICZ, A., AND SEBASTIANI, R. 2002. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *CADE*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 2392. Springer, 195–210.

BAADER, F. AND NIPKOW, T. 1998. *Term Rewriting and All That.* Cambridge University Press.

BAADER, F. AND SNYDER, W. 2001. Unification theory. See Robinson and Voronkov [2001], 445–533.

BACHMAIR, L. AND GANZINGER, H. 2001. Resolution theorem proving. See Robinson and Voronkov [2001], 19–99.

BACHMAIR, L., GANZINGER, H., LYNCH, C., AND SNYDER, W. 1992. Basic paramodulation and superposition. In *CADE*, D. Kapur, Ed. Lecture Notes in Computer Science, vol. 607. Springer, 462–476.

BACHMAIR, L., TIWARI, A., AND VIGNERON, L. 2003. Abstract congruence closure. *Journal of Automated Reasoning 31,* 2, 129–168.

BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation, 2001*. ACM Press, 203–313.

BARENDREGT, H. P. 1992. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds. Vol. 2. Oxford University Press, Chapter 2, 117–309.

BARNETT, M., DELINE, R., FÄHNDRICH, M., 0002, B. J., LEINO, K. R. M., SCHULTE, W., AND VENTER, H. 2005. The Spec# programming system: Challenges and directions. See Meyer and Woodcock [2008], 144–152.

BARRETT, C., DE MOURA, L., AND STUMP, A. 2005. SMT-COMP: Satisfiability modulo theories competition. In *Computer-Aided Verification, CAV '2005*, K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, Edinburgh, Scotland, 20–23.

BARRETT, C. AND TINELLI, C. 2007. CVC3. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer, 298–302.

BARRETT, C., TINELLI, C., SEBASTIANI, R., AND SESHIA, S. 2009. Satisfiability modulo theories. See Biere et al. [2009].

BARRETT, C. W., DILL, D. L., AND STUMP, A. 2002. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Computer-Aided Verification, CAV '02*. Lecture Notes in Computer Science. Springer-Verlag.

BARWISE, J., Ed. 1978a. *Handbook of Mathematical Logic*. Studies in Logic and the Foundations of Mathematics, vol. 90. North-Holland, Amsterdam, Holland.

BARWISE, J. 1978b. An introduction to first-order logic. See Barwise [1978a], Chapter A1, 5–46.

BASU, S., POLLACK, R., AND ROY, M.-F. 2003. *Algorithms in Real Algebraic Geometry*. Springer-Verlag.

BECKMAN, N., NORI, A. V., RAJAMANI, S. K., AND SIMMONS, R. J. 2008. Proofs from tests. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, B. G. Ryder and A. Zeller, Eds. ACM, 3–14.

BELINFANTE, J. G. F. 1999. Computer proofs in Gödel's class theory with equational definitions for composite and cross. *J. Autom. Reasoning 22,* 2, 311–339.

BENSALEM, S., LAKHNECH, Y., AND OWRE, S. 1998. InVeSt: A tool for the verification of invariants. See Hu and Vardi [1998], 505–510.

BEREZIN, S., GANESH, V., AND DILL, D. L. 2003. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, H. Garavel and J. Hatcliff, Eds. Lecture Notes in Computer Science, vol. 2619. Springer, 521–536.

BERTOT, Y. AND CASTÉRAN, P. 2004. *Interactive Theorem Proving and Program Development*. Springer. Coq home page: `http://coq.inria.fr/`.

BIERE, A. 2008. PicoSAT essentials. *JSAT 4,* 2-4, 75–97.

BIERE, A., CIMATTI, A., CLARKE, E., FUJITA, M., AND ZHU, Y. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. ACM Design Automation Conf. (DAC'99)*. ACM Press.

BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T., Eds. 2009. *Handbook of Satisfiability*. IOS Press.

BLACKBURN, P., DE RIJKE, M., AND VENEMA, Y. 2002. *Modal Logic*. Cambridge University Press.

BLEDSOE, W. W. AND BRUELL, P. 1974. A man-machine theorem-proving system. *Artificial Intelligence 5*, 51–72.

BOCKMAYR, A. AND WEISPFENNING, V. 2001. Solving numerical constraints. See Robinson and Voronkov [2001], 751–742.

BOFILL, M., NIEUWENHUIS, R., OLIVERAS, A., RODRÍGUEZ-CARBONELL, E., AND RUBIO, A. 2008. The Barcelogic SMT solver. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton,* *NJ, USA, July 7-14, 2008, Proceedings*, A. Gupta and S. Malik, Eds. Lecture Notes in Computer Science, vol. 5123. Springer, 294–298.

BOGUDLOV, I., LEV-AMI, T., REPS, T. W., AND SAGIV, M. 2007. Revamping TVLA: Making parametric shape analysis competitive. In *CAV*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer, 221–225.

BOOLOS, G. S. AND JEFFREY, R. C. 1989. *Computability and Logic*, third ed. Cambridge University Press, Cambridge, UK.

BÖRGER, E., GRÄDEL, E., AND GUREVICH, Y. 1997. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin.

BOROVANSKÝ, P., KIRCHNER, C., KIRCHNER, H., AND MOREAU, P.-E. 2002. ELAN from a rewriting logic point of view. *Theor. Comput. Sci 285,* 2, 155–185.

BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. 1975. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices 10,* 6 (June), 234–245.

BOYER, R. S., LUSK, E. L., MCCUNE, W., OVERBEEK, R. A., STICKEL, M. E., AND WOS, L. 1986. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning 2,* 3 (Sept.), 287–327.

BOYER, R. S. AND MOORE, J. S. 1975. Proving theorems about Lisp functions. *Journal of the ACM 22,* 1 (Jan.), 129–144.

BOYER, R. S. AND MOORE, J. S. 1979. *A Computational Logic*. Academic Press, New York, NY.

BOYER, R. S. AND MOORE, J. S. 1988. *A Computational Logic Handbook*. Academic Press, New York, NY.

BRADLEY, A. R. AND MANNA, Z. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag.

BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. 2006. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI*, E. A. Emerson and K. S. Namjoshi, Eds. Lecture Notes in Computer Science, vol. 3855. Springer, 427–442.

BRUTTOMESSO, R., CIMATTI, A., FRANZÉN, A., GRIGGIO, A., AND SEBASTIANI, R. 2008. The MathSAT 4 SMT solver. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, A. Gupta and S. Malik, Eds. Lecture Notes in Computer Science, vol. 5123. Springer, 299–303.

BRYANT, R. 1992. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24,* 3 (Sept.), 293–318.

BRYANT, R. E., LAHIRI, S. K., AND SESHIA, S. A. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification, CAV '2002*. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, Copenhagen, Denmark.

BUCHBERGER, B. 1976. A theoretical basis for the reduction of polynomials to canonical forms. *ACM SIGSAM Bulletin 10,* 3, 19–29.

BULTAN, T., GERBER, R., AND PUGH, W. 1997. Symbolic model checking of infinite state systems using Presburger arithmetic. See Grumberg [1997], 400–411.

BUNDY, A. 2001. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Vol. I. Elsevier Science, Chapter 13, 845–911.

BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98,* 2 (June), 142–170.

BURRIS, S. N. AND SANKAPPANAVAR, H. P. 1981. *A course in universal algebra*. Graduate Texts in Mathematics, vol. 78. Springer–Verlag. Revised edition online at `http://thoralf.uwaterloo.ca/htdocs/ualg.html`.

CARREÑO, V. AND MUÑOZ, C. 2000. Formal analysis of parallel landing scenarios. In *19th AIAA/IEEE Digital Avionics Systems Conference*. Philadelphia, PA.

CHURCH, A. 1936. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 345–363. Reprinted in [Davis 1965].

CHURCH, A. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic 5*, 56–68.

CLARKE, E. M., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50,* 5, 752–794.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. MIT Press.

CLARKE, L. A. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering 2,* 3 (Sept.), 215–222.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND QUESADA, J. F. 1999. The Maude system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, P. Narendran and M. Rusinowitch, Eds. Springer-Verlag LNCS 1631, Trento, Italy, 240–243.

CLAVEL, M., DURÁN, F., EKER, S., MESEGUER, J., AND STEHR, M.-O. 1999. Maude as a formal meta-tool. In *World Congress on Formal Methods (FM'99)*. Lecture Notes in Computer Science, vol. 1709. Springer-Verlag, 1684–1703.

COHEN, P. J. 1969. Decision procedures for real and p-adic fields. *Communications of Pure and Applied Mathematics 22,* 2, 131–151.

COLLINS, G. 1975. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings Second GI Conference on Automata Theory and Formal Languages*. Lecture Notes in Computer Science, vol. 33. Springer-Verlag, Berlin, 134–183.

CONSTABLE, R. L., ALLEN, S. F., BROMLEY, H. M., CLEAVELAND, W. R., CREMER, J. F., HARPER, R. W., HOWE, D. J., KNOBLOCK, T. B., MENDLER, N. P., PANANGADEN, P., SASAKI, J. T., AND SMITH, S. F. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ. Nuprl home page: `http://www.cs.cornell.edu/Info/Projects/NuPRL/`.

COOK, S. A. 1971. The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*. 151–158.

COOPER, D. C. 1972. Theorem proving in arithmetic without multiplication. In *Machine Intelligence 7*. Edinburgh University Press, 91–99.

COQUAND, T. AND HUET, G. 1988. The calculus of constructions. *Information and Computation 76,* 2/3, 95–120.

CORBETT, J., DWYER, M., HATCLIFF, J., PASAREANU, C., ROBBY, LAUBACH, S., AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*. IEEE Computer Society, Limerick, Ireland, 439–448.

COTTON, S. AND MALER, O. 2006. Fast and flexible difference constraint propagation for DPLL(T). In *SAT*, A. Biere and C. P. Gomes, Eds. Lecture Notes in Computer Science, vol. 4121. Springer, 170–183.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, Los Angeles, CA, 238–252.

CRAIG, W. 1957. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic 22,* 3, 269–285.

D'AGOSTINO, M., GABBAY, D. M., HÄNLE, R., AND POSEGGA, J., Eds. 1999. *Handbook of Tableau Methods*. Kluwer Academic Publishers, Dordrecht.

DALEN, D. V. 1983. *Logic and Structure*. Springer-Verlag.

DANTZIG, G. AND CURTIS, B. 1973. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory*, 288–297.

DARLINGTON, J. 1981. An experimental program transformation and synthesis system. *Artif. Intell. 16,* 1, 1–46.

DAVIS, M., Ed. 1965. *The Undecidable*. Raven Press, Hewlett, N.Y.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem proving. *Commun. ACM 5,* 7 (July), 394–397. Reprinted in Siekmann and Wrightson [Siekmann and Wrightson 1983], pages 267–270, 1983.

DAVIS, M. AND PUTNAM, H. 1960. A computing procedure for quantification theory. *Journal of the ACM 7,* 3, 201–215.

DE BRUIJN, N. G. 1970. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*. Lecture Notes in Mathematics, vol. 125. Springer-Verlag, Berlin, 29–61.

DE BRUIJN, N. G. 1980. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 589–606.

DE MOURA, L., DUTERTRE, B., AND SHANKAR, N. 2007. A tutorial on satisfiability modulo theories. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer-Verlag, 20–36.

DE MOURA, L., RUESS, H., AND SOREA, M. 2002. Lazy theorem proving for bounded model checking over infinite domains. In *18th International Conference on Automated Deduction (CADE)*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 2392. Springer-Verlag, Copenhagen, Denmark, 438–455.

DE MOURA, L. M. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, C. R. Ramakrishnan and J. Rehof, Eds. Lecture Notes in Computer Science, vol. 4963. Springer, 337–340.

DELZANNO, G. AND PODELSKI, A. 2001. Constraint-based deductive model checking. *STTT 3,* 3, 250–270.

DETLEFS, D., NELSON, G., AND SAXE, J. B. 2003. Simplify: A theorem prover for program checking. Tech. Rep. HPL-2003-148, HP Labs.

DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. 1998. Extended static checking. Tech. Rep. 159, COMPAQ Systems Research Center.

DOWEK, G. 2001. Higher-order unification and matching. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Vol. II. Elsevier Science, Chapter 16, 1009–1062.

DOWNEY, P. J. AND SETHI, R. 1978. Assignment commands with array references. *J. ACM 25,* 4, 652–666.

D'SILVA, V., KROENING, D., AND WEISSENBACHER, G. 2008. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems 27,* 7, 1165–1178.

DUTERTRE, B. AND DE MOURA, L. 2006a. A fast linear-arithmetic solver for DPLL(T). In *Computer-Aided Verification, CAV '2006*. Lecture Notes in Computer Science, vol. 4144. Springer-Verlag, Seattle, WA, 81–94.

DUTERTRE, B. AND DE MOURA, L. 2006b. The Yices SMT solver. `http://yices.csl.sri.com/`.

EBBINGHAUS, H.-D., FLUM, J., AND THOMAS, W. 1984. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, NY.

EÉN, N. AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *Proceedings of SAT 2003*.

EKER, S., LADEROUTE, K., LINCOLN, P., SRIRAM, M. G., AND TALCOTT, C. L. 2003. Representing and simulating protein functional domains in signal transduction using maude. In *Computational Methods in Systems Biology, First International Workshop, CMSB 2003, Roverto, Italy, February 24-26, 2003, Proceedings*, C. Priami, Ed. Lecture Notes in Computer Science, vol. 2602. Springer, 164–165.

ELGAARD, J., KLARLUND, N., AND MÖLLER, A. 1998. Mona 1.x: New techniques for WS1S and WS2S. See Hu and Vardi [1998], 516–520.

ELKADUWE, D., KLEIN, G., AND ELPHINSTONE, K. 2008. Verified protection model of the seL4 microkernel. In *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, N. Shankar and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 5295. Springer, 99–114.

ELSPAS, B., GREEN, M., MORICONI, M., AND SHOSTAK, R. 1979. A JOVIAL verifier. Tech. rep., Computer Science Laboratory, SRI International. Jan.

ELSPAS, B., LEVITT, K. N., WALDINGER, R. J., AND WAKSMAN, A. 1972. An assessment of techniques for proving program correctness. *ACM Comput. Surv. 4,* 2, 97–147.

EMERSON, E. A. 1990. Temporal and modal logic. See van Leeuwen [1990], Chapter 16, 995–1072.

ENDERTON, H. B. 1972. *A Mathematical Introduction to Logic*. Academic Press, New York, NY.

ESCOBAR, S., MEADOWS, C., AND MESEGUER, J. 2007. Equational cryptographic reasoning in the Maude-NRL protocol analyzer. *Electr. Notes Theor. Comput. Sci 171,* 4, 23–36.

FEFERMAN, S. 1978. Theories of finite type related to mathematical practice. See Barwise [1978a], Chapter D4, 913–972.

FEFERMAN, S. 2006. Tarski's influence on computer science. *Logical Methods in Computer Science 2,* 3:6, 1–13.

FILLIÂTRE, J.-C. AND MARCHÉ, C. 2007. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer, 173–177.

FISCHER, M. J. AND RABIN, M. O. 1974. Super-exponential complexity of presburger arithmetic. In *Complexity of computation*, R. M. Karp, Ed. American Mathematical Society, Providence, RI, 27–41.

FITTING, M. 1990. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag.

F.K. HANNA AND N. DAECHE. 1986. Specification and verification of digital systems using higher-order predicate logic. *IEE Proceedings 133 Part E,* 5 (Sept.), 242–254.

FLANAGAN, C., JOSHI, R., OU, X., AND SAXE, J. B. 2003. Theorem proving using lazy proof explication. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV 2003)*. Lecture Notes in Computer Science, vol. 2725. Springer-Verlag, 355–367.

FLOYD, R. W. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. American Mathematical Society, Providence, RI, 19–32.

FOURIER, J. B. J. 1826. Solution d'une question particulière du calcul des inégalités. *Nouveau Bulletin des Sciences par la Société philomathique de Paris*, 99–100.

FREGE, G. 1893–1903. *Grundgesetze der Arithmetik, Begriffsschriftlich abgeleitet*. Verlag Hermann Pohle, Jena.

GABBAY, D. M. AND GUENTHNER, F., Eds. 1983. *Handbook of Philosophical Logic–Volume I: Elements of Classical Logic*. Synthese Library, vol. 164. D. Reidel Publishing Company, Dordrecht, Holland.

GABBAY, D. M. AND GUENTHNER, F., Eds. 1984. *Handbook of Philosophical Logic–Volume II: Extensions of Classical Logic*. Synthese Library, vol. 165. D. Reidel Publishing Company, Dordrecht, Holland.

GABBAY, D. M. AND GUENTHNER, F., Eds. 1985. *Handbook of Philosophical Logic–Volume III: Alternatives to Classical Logic*. Synthese Library, vol. 166. D. Reidel Publishing Company, Dordrecht, Holland.

GALLER, B. A. AND FISHER, M. J. 1964. An improved equivalence algorithm. *Commun. ACM 7,* 5, 301–303.

GANZINGER, H. AND KOROVIN, K. 2006. Theory instantiation. In *LPAR*, M. Hermann and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 4246. Springer, 497–511.

GERHART, S. L., MUSSER, D. R., THOMPSON, D. H., BAKER, D. A., BATES, R. L., ERICKSON, R. W., LONDON, R. L., TAYLOR, D. G., AND WILE, D. S. 1980. An overview of Affirm: A specification and verification system. In *Information Processing '80*, S. H. Lavington, Ed. IFIP, North-Holland Publishing Company, Australia, 343–347.

GILMORE, P. C. 1960. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development 4*, 28–35. Reprinted in Siekmann and Wrightson [Siekmann and Wrightson 1983], pages 151–161, 1983.

GIRARD, J.-Y., LAFONT, Y., AND TAYLOR, P. 1989. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

GODEFROID, P., KLARLUND, N., AND SEN, K. 2005. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation: PLDI*. Association for Computing Machinery, Chicago, IL, 213–223.

GÖDEL, K. 1930. Über die vollständigkeit des logikkalküls. Ph.D. thesis, University of Vienna. Translated by Stefan Bauer-Mengelberg and reprinted in [van Heijenoort 1967, pages 582–591].

GÖDEL, K. 1967. On formally undecidable propositions of *principia mathematica* and related systems. First published 1930 and 1931.

GOGUEN, J., KIRCHNER, C., MEGRELIS, A., MESEGUER, J., AND WINKLER, T. 1987. An introduction to OBJ3. In *Conditional Term Rewriting Systems, 1st International workshop, Orsay, France*, S. Kaplan and J.-P. Jouannaud, Eds. Lecture Notes in Computer Science, vol. 308. Springer-Verlag, 258–263.

GOLDBERG, E. AND NOVIKOV, Y. 2007. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics 155,* 12, 1549–1561.

GOLDBLATT, R. 1992. *Logics of Time and Computation*, second ed. CSLI Lecture Notes, vol. 7. Center for the Study of Language and Information, Stanford, CA.

GOMES, C. P., KAUTZ, H., SABHARWAL, A., AND SELMAN, B. 2008. Satisfiability solvers. In *Handbook of Knowledge Representation*. Foundations of Artificial Intelligence, vol. 3. Elsevier, 89–134.

GONTHIER, G. 2008. Formal proof: The four-color theorem. *Notices of the American Mathematical Society 55,* 11 (Dec.), 1382–1394.

GOODSTEIN, R. L. 1964. *Recursive Number Theory*. North-Holland, Amsterdam.

GORDON, M. 1985a. HOL: A machine oriented formulation of higher order logic. Tech. Rep. 68, University of Cambridge Computer Laboratory, Cambridge, England. July.

GORDON, M. 1985b. Why higher-order logic is a good formalism for specifying and verifying hardware. Tech. Rep. 77, University of Cambridge Computer Laboratory. Sept.

GORDON, M. 1986. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, G. Milne and P. A. Subrahmanyam, Eds. Elsevier, 153–177. Reprinted in Yoeli [Yoeli 1990, pp. 57–77].

GORDON, M., MILNER, R., MORRIS, L., NEWEY, M., AND WADSWORTH, C. 1977. A metalanguage for interactive proof in LCF. Tech. Rep. CSR-16-77, Department of Computer Science, University of Edinburgh.

GORDON, M., MILNER, R., AND WADSWORTH, C. 1979. *Edinburgh LCF: A Mechanized Logic of Computation*. Lecture Notes in Computer Science, vol. 78. Springer-Verlag.

GORDON, M. J. C. 1989. Mechanizing programming logics in higher-order logic. In *Current Trends in Hardware Verification and Theorem Proving*, G. Birtwistle and P. A. Subrahmanyam, Eds. Springer-Verlag, New York, NY, 387–439.

GORDON, M. J. C. AND MELHAM, T. F., Eds. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK. HOL home page: `http://www.cl.cam.ac.uk/Research/HVG/HOL/`.

GREVE, D., WILDING, M., AND VANFLEET, W. 2003. A separation kernel formal security policy. In *Fourth International Workshop on the ACL2 Theorem Prover*. Boulder, CO. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/\#presentations`.

GRIES, D. AND SCHNEIDER, F. B. 1993. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY.

GRUMBERG, O., Ed. 1997. *Computer-Aided Verification, CAV '97*. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, Haifa, Israel.

GULWANI, S. AND TIWARI, A. 2006. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *European Symposium on Programming, ESOP 2006*, P. Sestoft, Ed. Lecture Notes in Computer Science, vol. 3924. Springer-Verlag, 279–293.

HALES, T. C. 2002. A computer verification of the Kepler conjecture. In *Proceedings of the International Congress of Mathematicians*.

HALES, T. C. 2007. The Jordan curve theorem, formally and informally. *AMM: The American Mathematical Monthly 114*.

HALES, T. C., HARRISON, J., MCLAUGHLIN, S., NIPKOW, T., OBUA, S., AND ZUMKELLER, R. 2009. A revision of the proof of the Kepler conjecture.

HALMOS, P. R. 1960. *Naive Set Theory*. The University Series in Undergraduate Mathematics. Van Nostrand Reinhold Company, New York, NY.

HALPERN, J. Y., HARPER, R., IMMERMAN, N., KOLAITIS, P. G., VARDI, M., AND VIANU, V. 2001. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic 7,* 2, 213–236.

HAMON, G., DE MOURA, L., AND RUSHBY, J. 2004. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, Beijing, China, 261–270.

HAMON, G., DE MOURA, L., AND RUSHBY, J. 2005. Automated test generation with SAL. Technical note, Computer Science Laboratory, SRI International, Menlo Park, CA. Sept. Available at `http://www.csl.sri.com/users/rushby/abstracts/sal-atg`.

HANTLER, S. L. AND KING, J. C. 1976. An introduction to proving the correctness of programs. *ACM Computing Surveys 8,* 3 (Sept.), 331–353.

HARPER, R., HONSELL, F., AND PLOTKIN, G. D. 1987. A framework for defining logics. In *IEEE Symposium on Logic in Computer Science*. Ithaca, NY.

HARRISON, J. 1996. HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas and A. Camilleri, Eds. Lecture Notes in Computer Science, vol. 1166. Springer-Verlag, Palo Alto, CA, 265–269. HOL Light home page: `http://www.cl.cam.ac.uk/~jrh13/hol-light/index.html`.

HARRISON, J. 2001. The LCF approach to theorem proving. Available from `http://www.cl.cam.ac.uk/~jrh13/slides/manchester-12sep01/slides.pdf`.

HARRISON, J. 2006. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006* (May 2006), M. Bernardo and A. Cimatti, Eds. Lecture Notes in Computer Science, vol. 3965. Springer-Verlag, Bertinoro, Italy, 211–242.

HARRISON, J. 2009. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

HENKIN, L. 1949. The completeness of first-order functional calculus. *Journal of Symbolic Logic 14,* 3, 159–166.

HENKIN, L. 1950. Completeness in the theory of types. *Journal of Symbolic Logic 15,* 2 (June), 81–91.

HENKIN, L. 1996. The discovery of my completeness proofs. *BSL: The Bulletin of Symbolic Logic 2,* 2, 127–158.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2003. Software verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*. Lecture Notes in Computer Science, vol. 2648. Springer-Verlag, 235–239. BLAST home page: `http://embedded.eecs.berkeley.edu/blast/`.

HERBRAND, J. 1930. Recherches sur la théorie de la démonstration. Ph.D. thesis, Université de Paris, Paris, France. English translation published in van Heijenoort [1967] and Herbrand [1971].

HERBRAND, J. 1971. *Logical Writings*. Harvard University Press.

HIERONS, R. M., BOGDANOV, K., BOWEN, J. P., CLEAVELAND, R., DERRICK, J., DICK, J., GHEORGHE, M., HARMAN, M., KAPOOR, K., KRAUSE, P., LÜTTGEN, G., SIMONS, A. J. H., VILKOMIR, S., WOODWARD, M. R., AND ZEDAN, H. 2009. Using formal specifications to support testing. *ACM Comput. Surv. 41,* 2, 1–76.

HILBERT, D. 1902. Mathematical problems. Lecture delivered before the International Congress of Mathematicians at Paris in 1900. *Bulletin of the American Mathematical Society 8*, 437–479.

HOARE, C. A. R. 2003. The verifying compiler: A grand challenge for computing research. *Journal of the ACM 50,* 1, 63–69.

HOARE, C. A. R. AND MISRA, J. 2008. Verified software: Theories, tools, experiments vision of a grand challenge project. In *Verified Software: Theories, Tools, Experiments*, B. Meyer and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 4171. Springer-Verlag.

HODGES, W. 1997. *A Shorter Model Theory*. Cambridge University Press.

HÖRMANDER, L. 1983. *The analysis of linear partial differential operators II: Differential operators with constant coefficients*. Grundlehren der math. Wissenschaften, vol. 257. Springer.

HOWARD, W. 1980. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 479–490.

HU, A. J. AND VARDI, M. Y., Eds. 1998. *Computer-Aided Verification, CAV '98*. Lecture Notes in Computer Science, vol. 1427. Springer-Verlag, Vancouver, Canada.

HUTH, M. R. A. AND RYAN, M. D. 2000. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, UK.

JACKSON, D. 2006. *Software Abstractions: Logic, Language, and Analysis*. MIT Press.

JACKSON, P. AND SHERIDAN, D. 2004. Clause form conversions for boolean circuits. In *SAT (Selected Papers)*, H. H. Hoos and D. G. Mitchell, Eds. Lecture Notes in Computer Science, vol. 3542. Springer, 183–198.

JHALA, R. AND MAJUMDAR, R. 2009. Software model checking. *ACM Computing Surveys*. This volume.

JONES, C. B. 1990. *Systematic Software Development Using VDM*, second ed. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK.

JONES, C. B. 1992. The search for tractable ways of reasoning about programs. Tech. Rep. UMCS-92-4-4, Department of Computer Science, University of Manchester, Manchester, UK. Mar.

KAUFMANN, M., MANOLIOS, P., AND MOORE, J. S. 2000. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods, vol. 3. Kluwer.

KAUFMANN, M. AND MOORE, J. S. 1996. ACL2: An industrial strength version of Nqthm. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*. IEEE Washington Section, Gaithersburg, MD, 23–34.

KAUTZ, H. AND SELMAN, B. 1996. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*. Wiley, 359–363.

KESTEN, Y. AND PNUELI, A. 1998. Modularization and abstraction: The keys to practical formal verification. In *Mathematical Foundations of Computer Science*. 54–71.

KING, J. C. 1969. A program verifier. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

KING, J. C. 1976. Symbolic execution and program testing. *CACM 19,* 7, 385–394.

KING, J. C. AND FLOYD, R. W. 1970. An interpretation oriented theorem prover over integers. In *STOC '70: Proceedings of the second annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 169–179.

KLEENE, S. C. 1952. *Introduction to Metamathematics*. North-Holland, Amsterdam, The Netherlands.

KLEENE, S. C. 1967. *Mathematical Logic*. John Wiley and Sons, New York, NY.

KNAPP, S. AND PAUL, W. 2007. Pervasive verification of distributed real-time systems. In *Software System Reliability and Security*, M. Broy, J. Grünbauer, and T. Hoare, Eds. IOS Press, NATO Security Through Science Series. Sub-Series D: Information and Communication Security, vol. 9. 239–297.

KOZEN, D. 1977. Complexity of finitely presented algebras. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*. Boulder, Colorado, 164–177.

KROENING, D., CLARKE, E., AND YORAV, K. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*. ACM Press, 368–371.

KRÖNING, D. AND STRICHMAN, O. 2008. *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag.

KUNEN, K. 1980. *Set Theory: An Introduction to Independence Proofs*. Studies in Logic and the Foundations of Mathematics, vol. 102. North-Holland, Amsterdam, The Netherlands.

LEIVANT, D. 1994. Higher order logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Deduction Methodologies*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Clarendon Press, Oxford, 229–321.

LEROY, X. 2007. Formal verification of an optimizing compiler. In *MEMOCODE*. IEEE, 25.

LEVIN, L. 1973. Universal search problems. *Problemy Peredachi Informatsii 9,* 3, 265–266. English translation in Trakhtenbrot, B. A.: A survey of Russian approaches to Perebor (brute-force search) algorithms. Annals of the History of Computing, 6 (1984), pages. 384-400.

LIU, X., KREITZ, C., VAN RENESSE, R., HICKEY, J., HAYDEN, M., BIRMAN, K. P., AND CONSTABLE, R. L. 1999. Building reliable, high-performance communication systems from components. In *SOSP*. 80–92.

LUCKHAM, D. C., GERMAN, S. M., VON HENKE, F. W., KARP, R. A., MILNE, P. W., OPPEN, D. C., POLAK, W., AND SCHERLIS, W. L. 1979. Stanford Pascal Verifier user manual. CSD Report STAN-CS-79-731, Stanford University, Stanford, CA. Mar.

LUO, Z. AND POLLACK, R. 1992. The LEGO proof development system: A user's manual. Tech. Rep. ECS-LFCS-92-211, University of Edinburgh.

MANNA, Z., STICKEL, M., AND WALDINGER, R. 1991. Monotonicity properties in automated deduction. In *Artificial Intelligence and Mathematical Theorem of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed. Academic Press, 261–280.

MANNA, Z. AND WALDINGER, R. A deductive approach to program synthesis. *ACM Trans. on Prog. Langs. and Sys. 2,* 1.

MANOLIOS, P. AND VROON, D. 2007. Efficient circuit to CNF conversion. In *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, J. Marques-Silva and K. A. Sakallah, Eds. Lecture Notes in Computer Science, vol. 4501. Springer, 4–9.

MARQUES-SILVA, J. AND SAKALLAH, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48,* 5 (May), 506–521.

MARTIN-LÖF, P. 1980. *Intuitionistic type theory*. Bibliopolis, Napoli.

MATIYASEVICH, Y. V. 1993. *Hilbert's Tenth Problem*. MIT Press, Cambridge, Massachusetts.

MCCARTHY, J. 1962. Computer programs for checking mathematical proofs. In *Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics*. Vol. V. American Mathematical Society, Providence, Rhode Island, 219–227.

MCCARTHY, J. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hershberg, Eds. North-Holland.

MCCUNE, W. 1997. Solution of the robbins problem. *Journal of Automated Reasoning 19,* 3 (Dec.), 263–276.

MCCUNE, W. W. 1990. OTTER 2.0 users guide. Tech. Rep. ANL-90/9, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL. Mar.

MCCUNE, W. W. 2007. The Prover9 reference manual.

MCMILLAN, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA.

MCMILLAN, K. L. 2003. Interpolation and SAT-based model checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, W. A. H. Jr. and F. Somenzi, Eds. Lecture Notes in Computer Science, vol. 2725. Springer, 1–13.

MENDELSON, E. 1964. *Introduction to Mathematical Logic*. The University Series in Undergraduate Mathematics. D. Van Nostrand Company, New York, NY.

MESEGUER, J. 1989. General logics. In *Logic Colloquium '87*. North Holland, Amsterdam, 275–329.

MESEGUER, J. AND ROSU, G. 2005. Computational logical frameworks and generic program analysis technologies. See Meyer and Woodcock [2008], 256–267.

MEYER, B. AND WOODCOCK, J., Eds. 2008. *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. Lecture Notes in Computer Science, vol. 4171. Springer.

MILLEN, J. AND RUESS, H. 2000. Protocol-independent secrecy. In *Proceedings of the Symposium on Security and Privacy*, M. Reiter and R. Needham, Eds. IEEE Computer Society, Oakland, CA, 110–119.

MILLER, D. AND NADATHUR, G. 1986. Higher-order logic programming. In *IEEE Symposium on Logic Programming*.

MILNER, R. 1972. Logic for computable functions: description of a machine implementation. Technical Report CS-TR-72-288, Stanford University, Department of Computer Science. May.

MINER, P. S., GESER, A., PIKE, L., AND MADDALON, J. 2004. A unified fault-tolerance protocol. In *Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, Y. Lakhnech and S. Yovine, Eds. Lecture Notes in Computer Science, vol. 3253. Springer, 167–182.

MINTS, G. 1992. *A Short Introduction to Modal Logic*. CSLI Lecture Notes, vol. 30. Center for the Study of Language and Information, Stanford, CA.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*. 530–535.

NADATHUR, G. AND MILLER, D. 1990. Higher-order Horn clauses. *Journal of the ACM 37,* 4, 777–814.

NADATHUR, G. AND MITCHELL, D. J. 1999. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In *16th Conference on Automated Deduction (CADE)*, H. Ganzinger, Ed. Number 1632 in LNAI. Springer, Trento, 287–291.

NAUR, P. 1966. Proof of algorithms by general snapshots. *BIT 6*, 310–316.

NEDERPELT, R. P., GEUVERS, J. H., AND DE VRIJER, R. C. 1994. *Selected Papers on Automath*. North-Holland, Amsterdam.

NELSON, G. 1981. Techniques for program verification. Tech. Rep. CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca.

NELSON, G. AND OPPEN, D. 1977. Fast decision algorithms based on congruence closure. Tech. Rep. STAN-CS-77-646, Computer Science Department, Stanford University.

NELSON, G. AND OPPEN, D. C. 1979. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst. 1,* 2, 245–257.

NIEUWENHUIS, R. AND OLIVERAS, A. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05 (Nara, Japan)*, J. Giesl, Ed. Lecture Notes in Computer Science, vol. 3467. Springer, 453–468.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM 53,* 6, 937–977.

NIEUWENHUIS, R. AND RUBIO, A. 1992. Basic superposition is complete. In *ESOP*, B. Krieg-Brückner, Ed. Lecture Notes in Computer Science, vol. 582. Springer, 371–389.

NIEUWENHUIS, R. AND RUBIO, A. 2001. Paramodulation-based theorem proving. See Robinson and Voronkov [2001], 371–443.

NIPKOW, T., PAULSON, L. C., AND WENZEL, M. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer. Isabelle home page: `http://isabelle.in.tum.de/`.

O'CONNOR, R. 2005. Essential incompleteness of arithmetic verified by Coq. *CoRR abs/cs/0505034*. informal publication.

OHLBACH, H., NONNENGART, A., DE RIJKE, M., AND GABBAY, D. 2001. Encoding two-valued nonclassical logics in classical logic. See Robinson and Voronkov [2001], 1403–1486.

OPPEN, D. C. 1980. Complexity, convexity and combinations of theories. *Theoretical Computer Science 12*, 291–302.

OWRE, S., RUSHBY, J., SHANKAR, N., AND VON HENKE, F. 1995. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering 21,* 2 (Feb.), 107–125. PVS home page: `http://pvs.csl.sri.com`.

PARK, D. 1976. Finiteness is mu-ineffable. *Theoretical Computer Science 3*, 173–181.

PAULIN-MOHRING, C. 1993. Inductive definitions in the system Coq: Rules and properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, M. Bezem and J. F. Groote, Eds. Springer-Verlag LNCS 664, Utrecht, The Netherlands, 328–345.

PAULSON, L. 1998. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security 6,* 1, 85–128.

PAULSON, L. C. 1994. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science, vol. 828. Springer-Verlag. Isabelle home page: `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

PAULSON, L. C. 2003. The relative consistency of the axiom of choice mechanized using Isabelle/ZF. *LMS Journal of Computation and Mathematics 6*, 198–248.

PFENNING, F. 2001. Logical frameworks. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Vol. II. Elsevier Science, Chapter 17, 1063–1147.

PFENNING, F. AND SCHÜRMANN, C. 1999. Twelf - a meta-logical framework for deductive systems (system description). In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, H. Ganzinger, Ed. Lecture Notes in Artificial Intelligence, vol. 1632. Springer-Verlag, 202–206.

PNUELI, A. AND SHAHAR, E. 1996. A platform for combining deductive with algorithmic verification. In *8th International Computer Aided Verification Conference*. 184–195.

PRAWITZ, D. 1960. An improved proof procedure. *Theoria 26*, 102–139. Reprinted in Siekmann and Wrightson [Siekmann and Wrightson 1983], pages 162–201, 1983.

PUGH, W. 1992. A practical algorithm for exact array dependence analysis. *Communications of the ACM 35,* 8, 102–114.

QUAIFE, A. 1992. Automated deduction in von Neumann-Bernays-Gödel set theory. *Journal of Automated Reasoning 8,* 1 (Feb.), 91–147.

RABIN, M. O. 1978. Decidable theories. See Barwise [1978a], Chapter C8, 595–629.

RAJAN, S., SHANKAR, N., AND SRIVAS, M. 1995. An integration of model-checking with automated proof checking. In *Computer-Aided Verification (CAV) 1995, Liege, Belgium, Lecture Notes in Computer Science, Volume 939*. Springer Verlag, 84–97.

RIAZANOV, A. AND VORONKOV, A. 2002. The design and implementation of VAMPIRE. *AI Communications: Special issue on CASC 15,* 2 (Sept.), 91–110.

ROBINSON, A. AND VORONKOV, A., Eds. 2001. *Handbook of Automated Reasoning*. Elsevier Science.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *12,* 1, 23–41.

ROBINSON, L., LEVITT, K. N., AND SILVERBERG, B. A. 1979. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, CA. Three Volumes.

RUDNICKI, P. 1992. An overview of the MIZAR project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Båstad, Sweden, 311–330. The complete proccedings are available at `http://www.cs.chalmers.se/pub/cs-reports/baastad.92/`; this particular paper is also available separately at `http://web.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps`.

RUESS AND SHANKAR, N. 2004. Solving linear arithmetic constraints. Tech. Rep. CSL-SRI-04-01, SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA, 94025. January. revised, August 2004.

RUESS, H., SHANKAR, N., AND SRIVAS, M. K. 1996. Modular verification of SRT division. In *Computer-Aided Verification, CAV '96*, R. Alur and T. A. Henzinger, Eds. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, New Brunswick, NJ, 123–134.

RUSSELL, B. 1903. *The Principles of Mathematics*. Cambridge University Press.

RUSSELL, B. 1908. Mathematical logic based on the theory of types. *American Journal of Mathematics 30*, 222–262.

RUSSINOFF, D. M. 1999. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in Systems Design 14,* 1 (Jan.), 75–125.

RYAN, L. 2004. Efficient algorithms for clause-learning SAT solvers. M.S. thesis, Simon Fraser University.

SAALTINK, M. 1997. The Z/EVES system. In *ZUM '97: The Z Formal Specification Notation; 10th International Conference of Z Users*. Lecture Notes in Computer Science, vol. 1212. Springer-Verlag, Reading, UK, 72–85.

SAÏDI, H. AND GRAF, S. 1997. Construction of abstract state graphs with PVS. See Grumberg [1997], 72–83.

SAÏDI, H. AND SHANKAR, N. 1999. Abstract and model check while you prove. In *Computer-Aided Verification, CAV '99*, N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, Trento, Italy, 443–454.

SCHULZ, S. 2002. E – A Brainiac Theorem Prover. *Journal of AI Communications 15,* 2/3, 111–126.

SCOTT, D. S. 1993. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci 121,* 1&2, 411–440. Typed notes circulated in 1969.

SHANKAR, N. 2005. Inference systems for logical algorithms. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, R. Ramanujam and S. Sen, Eds. Lecture Notes in Computer Science, vol. 3821. Springer Verlag, 60–78.

SHANKAR, N. AND RUESS, H. 2002. Combining Shostak theories. In *International Conference on Rewriting Techniques and Applications (RTA '02)*, S. Tison, Ed. Lecture Notes in Computer Science, vol. 2378. Springer-Verlag, Copenhagen, Denmark, 1–18.

SHEERAN, M., SINGH, S., AND STÅLMARCK, G. 2000. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design (FMCAD 2000)*, W. A. Hunt, Jr. and S. D. Johnson, Eds. Lecture Notes in Computer Science, vol. 1954. Springer-Verlag, Austin, TX, 108–125.

SHOENFIELD, J. R. 1967. *Mathematical Logic*. Addison-Wesley, Reading, MA.

SHOSTAK, R. E. 1978. An algorithm for reasoning about equality. *Commun. ACM 21,* 7 (July), 583–585.

SHOSTAK, R. E. 1984. Deciding combinations of theories. *Journal of the ACM 31,* 1 (Jan.), 1–12.

SHOSTAK, R. E., SCHWARTZ, R., AND MELLIAR-SMITH, P. M. 1982. STP: A mechanized logic for specification and verification. In *6th International Conference on Automated Deduction (CADE)*, D. Loveland, Ed. Lecture Notes in Computer Science, vol. 138. Springer-Verlag, New York, NY.

SIEKMANN, J. AND WRIGHTSON, G., Eds. 1983. *Automation of Reasoning: Classical Papers on Computational Logic, Volumes 1 & 2*. Springer-Verlag.

SIMON, L., BERRE, D. L., AND HIRSCH, E. A. 2005. The SAT2002 competition. *Ann. Math. Artif. Intell 43,* 1, 307–342.

SKOLEM, T. 1967. The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains. See van Heijenoort [1967], 302–333. First published 1923.

SKOLEM, T. A. 1962. *Abstract Set Theory*. Number 8 in Notre Dame Mathematical Lectures. University of Notre Dame Press, Notre Dame, IN.

SLIND, K. AND NORRISH, M. 2008. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, O. A. Mohamed, C. Muñoz, and S. Tahar, Eds. Lecture Notes in Computer Science, vol. 5170. Springer, 28–32.

SMITH, D. R. 1990. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering 16,* 9 (September), 1024–1043.

SMITH, M. K., GOOD, D. I., AND DI VITO, B. L. 1988. Using the Gypsy methodology. Tech. Rep. 1, Computational Logic Inc. Jan.

SMULLYAN, R. M. 1968. *First-Order Logic*. Springer-Verlag. Republished by Courier Dover Publications, 1995.

SPIVEY, J. M., Ed. 1993. *The Z Notation: A Reference Manual*, second ed. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK.

STEELE JR., G. L. 1990. *Common Lisp: The Language*, second ed. Digital Press, Bedford, MA.

STICKEL, M. E. 1985. Automated deduction by theory resolution. *Journal of Automated Reasoning 1,* 4 (Dec.), 333–355.

STICKEL, M. E., WALDINGER, R. J., AND CHAUDHRI, V. K. 2000. A guide to SNARK.

STUMP, A., BARRETT, C. W., AND DILL, D. L. 2002. CVC: A cooperating validity checker. In *Proc. of CAV'02*. LNCS, vol. 2404.

STUMP, A., BARRETT, C. W., DILL, D. L., AND LEVITT, J. R. 2001. A decision procedure for an extensional theory of arrays. In *LICS*. 29–37.

SUPPES, P. 1972. *Axiomatic Set Theory*. Dover Publications, Inc., New York, NY.

SUTCLIFFE, G. AND SUTTNER, C. B. 2006. The state of CASC. *AI Commun 19,* 1, 35–48.

SZABO, M. E., Ed. 1969. *The Collected Papers of Gerhard Gentzen*. North-Holland.

TARJAN, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM 22,* 2, 215–225.

TARSKI, A. 1948. *A Decision Method for Elementary Algebra and Geometry*. University of California Press.

TARSKI, A., MOSTOWSKI, A., AND ROBINSON, R. M. 1971. *Undecidable Theories*. North-Holland, Amsterdam.

THE COQ DEVELOPMENT TEAM. 2009. The Coq proof assistant reference manual version 8.2. Tech. rep., INRIA. Feb.

THÉRY, L. 1998. A certified version of Buchberger's algorithm. In *Proceedings of CADE-15*, H. Kirchner and C. Kirchner, Eds. Number 1421 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, Germany, 349–364.

TIWARI, A. 2005. An algebraic approach for the unsatisfiability of nonlinear constraints. In *Computer Science Logic, 14th Annual Conf., CSL 2005*, L. Ong, Ed. Lecture Notes in Computer Science, vol. 3634. Springer-Verlag, 248–262.

TORLAK, E. AND JACKSON, D. 2007. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007*, O. Grumberg and M. Huth, Eds. Lecture Notes in Computer Science, vol. 4424. Springer, 632–647.

TROELSTRA, A. AND VAN DALEN, D. 1988. *Constructivity in Mathematics*. North-Holland, Amsterdam.

TSEITIN, G. 1968. On the complexity of derivation in propositional calculus. 115–125. Reprinted in J. Siekmann, and G. Wrightson [Siekmann and Wrightson 1983].

TURING, A. M. 1965. On computable numbers, with an application to the Entscheidungsproblem. See Davis [1965], 116–154. First published 1937.

VAN BENTHEM, J. AND DOETS, K. 1983. Higher-order logic. In *Handbook of Philosophical Logic, Volume I: Elements of Classical Logic*, D. Gabbay and F. Guenthner, Eds. Synthese Library, vol. 164. D. Reidel Publishing Co., Dordrecht, Chapter I.4, 275–329.

VAN DEN BERG, J. AND JACOBS, B. 2001. The LOOP compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, T. Margaria and W. Yi, Eds. Lecture Notes in Computer Science, vol. 2031. Springer-Verlag, Genova, Italy, 299–312.

VAN HEIJENOORT, J., Ed. 1967. *From Frege to Gödel: A Sourcebook of Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA.

VAN LEEUWEN, J., Ed. 1990. *Handbook of Theoretical Computer Science*. Vol. B: Formal Models and Semantics. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA.

VANDERBEI, R. 2001. *Linear Programming: Foundations and Extensions*. Kluwer's International Series.

WANG, C., IVANČIĆ, F., GANAI, M., AND GUPTA, A. 2005. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *Proceedings of International Conference on Logic for Artificial Intelligence and Reasoning (LPAR)*. Number 3835 in Lecture Notes in Artificial Intelligence. 322–336.

WEIDENBACH, C., BRAHM, U., HILLENBRAND, T., KEEN, E., THEOBALT, C., AND TOPIC, D. 2002. SPASS version 2.0. In *Automated Deduction – CADE-18*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 2392. Springer-Verlag, 275–279.

WELD, D. AND WOLFMAN, S. 1999. The LPSAT system and its application to resource planning. In *Proceedings 16th International Joint Conference on Artificial Intelligence (IJCAI)*. Stockholm, Sweden.

WENZEL, M. 1999. Isar - A generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds. Lecture Notes in Computer Science, vol. 1690. Springer, 167–184.

WEYHRAUCH, R. W. 1980. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence 13,* 1 and 2 (Apr.), 133–170.

WEYHRAUCH, R. W. AND THOMAS, A. J. 1974. FOL: A proof checker for first order logic. Tech. Rep. AIM-235, Stanford University, Computer Science Department, Artificial Intelligence Laboratory.

WIES, T., KUNCAK, V., ZEE, K., PODELSKI, A., AND RINARD, M. C. 2006. On verifying complex properties using symbolic shape analysis. *CoRR abs/cs/0609104*. informal publication.

WILLIAMS, H. P. 1976. Fourier-Motzkin elimination extension to integer programming problems. *J. Comb. Theory, Ser. A 21,* 1, 118–123.

YOELI, M., Ed. 1990. *Formal Verification of Hardware Design*. IEEE Computer Society, Los Alamitos, CA.

ZERMELO, E. 1908. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen 59*, 261–281. Translation by van Heijenoort in *From Frege to Gödel*.

ZHANG, H. 1997. SATO: An efficient propositional prover. In *Conference on Automated Deduction*. 272–275.

ZHANG, L. AND MALIK, S. 2002. The quest for efficient boolean satisfiability solvers. In *Proceedings of CADE-19*, A. Voronkov, Ed. Springer-Verlag, Berlin, Germany.

ZHANG, L. AND MALIK, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*. IEEE Computer Society, 10880–10885.