Verification: Industrial Applications
Notes to accompany lectures at 2003 Marktoberdorf Summer School

John Harrison
Intel Corporation, JF1-13
2111 NE 25th Avenue
Hillsboro OR, USA
johnh@ichips.intel.com

3 July 2003

Abstract

These lectures are intended to give a broad overview of the most important formal verification techniques that are currently used in the hardware industry. They are somewhat biased towards applications of deductive theorem proving (since that is my special area of interest) and away from temporal logic model checking (since there are other lectures on that topic). The arrangement of material is roughly in order of logical complexity, starting with methods for propositional logic and leading up to general theorem proving, then finishing with an extended case study on the verification of a floating-point square root algorithm used by Intel.

1. Propositional logic
2. Symbolic simulation
3. Model checking
4. General theorem proving
5. Case study of floating-point verification

The treatment of the various topics is quite superficial, with the aim being overall perspective rather than in-depth understanding. The last lecture is somewhat more detailed and should give a good feel for the realities of formal verification in this field.

These notes draw on my forthcoming book on automated theorem proving, and simple-minded example code (written in Objective Caml) illustrating many of the techniques described can be found on my Web page:

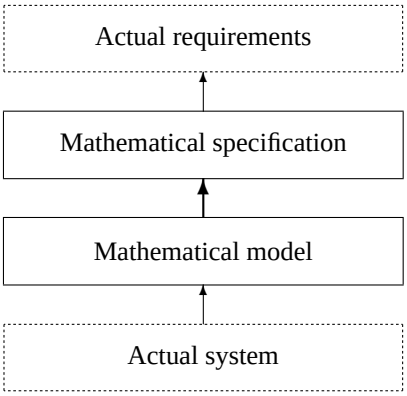
http://www.cl.cam.ac.uk/users/jrh/atp/index.html

0 Introduction

As most programmers know to their cost, writing programs that function correctly in all circumstances — or even saying what that means — is difficult. Most large programs contain 'bugs'. In the past, hardware has been substantially simpler than software, but this difference is eroding, and current leading-edge microprocessors are also extremely complex and usually contain errors. It has often been noted that mere testing, even on clever sets of test cases, is usually inadequate to guarantee correctness on *all* inputs, since the number of possible inputs and internal states, while finite, is usually astronomically large. For example [38]:

> As I have now said many times and written in many places: program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.

The main alternative to testing is formal verification, where it is rigorously *proved* that the system functions correctly on all possible inputs. This involves forming mathematical models of the system and its intended behaviour and linking the two:



Hardware versus software

The facts that (i) getting a mathematical proof right is also difficult [37], and (ii) correctness of formal models does not necessarily imply correctness of the actual system [41] caused much pointless controversy in the 70s. It is now widely accepted that formal verification, with the proof itself checked by machine, gives a much greater degree of confidence than traditional techniques.

The main impediment to greater use of formal verification is not these generalist philosophical objections, but just the fact that it's rather difficult. Only in a few isolated safety-critical niches of the software industry is any kind of formal verification widespread, e.g. in avionics. But in the hardware industry, formal verification is widely practised, and increasingly seen as necessary. We can identify at least three reasons:

- Hardware is designed in a more modular way than most software. Constraints of interconnect layering and timing means that one cannot really design 'spaghetti hardware'.

- More proofs in the hardware domain can be largely automated, reducing the need for intensive interaction by a human expert with the mechanical theorem-proving system.
- The potential consequences of a hardware error are greater, since such errors often cannot be patched or worked around, and may *in extremis* necessitate a hardware replacement.

To emphasize the last point, an error in the FDIV (floating-point division) instruction of some early Intel Intel® Pentium® processors resulted in 1994/5 in a charge to Intel of approximately $500M. Given this salutary lesson, and the size and diversity of its market, it's therefore understandable that Intel is particularly interested in formal verification.

The spectrum of formal verification techniques

There are a number of different formal verification techniques used in the hardware industry. Roughly speaking, these use different logics to achieve a particular balance between generality and automation. At one end of the scale, propositional logic has limited expressiveness, but (amazingly) efficient decision methods are available. At the other end, higher-order logic is very expressive but no efficient decision methods are available (in fact, higher-order validity is not even recursively enumerable) and therefore proofs must be constructed interactively with extensive user guidance. The lectures that follow are organized on the same basis, with propositional logic first, followed by symbolic simulation and temporal logic model checking, and finally general theorem proving at the end.

The particular method that is most appropriate for a given task may depend on details of that application. Needless to say, the effective combination of all methods is attractive, and there has been much interest in this idea [95, 64, 89]. Another 'hybrid' application of formal verification is to prove correctness of some of the CAD tools used to produce hardware designs [1] or even of the abstraction and reduction algorithms used to model-check large or infinite-state systems [23].

There are at least two books [69, 83] that also aim at a broad overview of formal verification techniques as applied to hardware and software, and are a useful complement to the material to follow.

1 Propositional logic

I assume that everyone knows what propositional logic is. The following is mainly intended to establish notation. Propositional logic deals with basic assertions that may be either true or false and various ways of combining them into composite propositions, without analyzing their internal structure. It is very similar to Boole's original algebra of logic, and for this reason the field is sometimes called *Boolean algebra* instead. In fact, the Boolean notation for propositional operators is still widely used by circuit designers.

English	Standard	Boolean	Other
false	$\bot$	0	F
true	$\top$	1	T
not p	$\neg p$	$\overline{p}$	$-p$, $\sim p$
p and q	$p \wedge q$	pq	$p\&q$, $p \cdot q$
p or q	$p \vee q$	$p + q$	$p \mid q$, p *or* q
p implies q	$p \Rightarrow q$	$p \leq q$	$p \rightarrow q$, $p \supset q$
p iff q	$p \Leftrightarrow q$	$p = q$	$p \equiv q$, $p \sim q$

For example, $p \wedge q \Rightarrow p \vee q$ means 'if p and q are true, then p or q is true'. We assume that 'or' is interpreted inclusively, i.e. $p \vee q$ means 'p or q or both'. This formula is therefore always true, or a *tautology*.

Logic and circuits

At a particular time-step, we can regard each internal or external wire in a (binary) digital computer as having a Boolean value, 'false' for 0 and 'true' for 1, and think of each circuit element as a Boolean function, operating on the values on its input wire(s) to produce a value at its output wire. The most basic building-blocks of computers used by digital designers, so-called *logic gates*, correspond closely to the usual logical connectives. For example an 'AND gate' is a circuit element with two inputs and one output whose output wire will be high (true) precisely if both the input wires are high, and so it corresponds exactly in behaviour to the 'and' ('$\wedge$') connective. Similarly a 'NOT gate' (or *inverter*) has one input wire and one output wire, which is high when the input is low and low when the input is high; hence it corresponds to the 'not' connective ('$\neg$'). Thus, there is a close correspondence between digital circuits and formulas which can be sloganized as follows:

Digital design	Propositional Logic
circuit	formula
logic gate	propositional connective
input wire	atom
internal wire	subexpression
voltage level	truth value

An important issue in circuit design is proving that two circuits have the same function, i.e. give identical results on all inputs. This arises, for instance, if a designer makes some special optimizations to a circuit and wants to check that they are "safe". Using the above correspondences, we can translate such problems into checking that a number of propositional formulas $P_n \Leftrightarrow P'_n$ are tautologies. Slightly more elaborate problems in circuit design (e.g. ignoring certain 'don't care' possibilities) can also be translated to tautology-checking. Thus, efficient methods for tautology checking directly yield useful tools for hardware verification.

Tautology checking

Tautology checking is apparently a difficult problem in general: it is co-NP complete, the dual problem 'SAT' of propositional satisfiability being the original NP-complete

problem [28]. However, there are a variety of algorithms that often work well on the kinds of problems arising in circuit design. And not just circuit design. The whole point of NP completeness is that many other apparently difficult combinatorial problems can be reduced to tautology/satisfiability checking. Recently it's become increasingly clear that this is useful not just as a theoretical reduction but as a practical approach. Surprisingly, many combinatorial problems are solved better by translating to SAT than by customized algorithms! This probably reflects the enormous engineering effort that has gone into SAT solvers.

The simplest method of tautology checking is, given a formula with n primitive propositional variables, to try all 2^n possible combinations and see if the formula always comes out true. Obviously, this may work for small n but hardly for the cases of practical interest. More sophisticated SAT checkers, while still having runtimes exponential in n in the worst case, do very well on practical problems that may even involve *millions* of variables. These still usually involve true/false case-splitting of variables, but in conjunction with more intelligent simplification.

The Davis-Putnam method

Most high-performance SAT checkers are based on the venerable Davis-Putnam algorithm [36], or more accurately on the 'DPLL' algorithm, a later improvement [35].

The starting-point is to put the formula to be tested for satisfiability in 'conjunctive normal form'. A formula is said to be in conjunctive normal form (CNF) when it is an 'and of ors', i.e. of the form:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

with each C_i in turn of the form:

$$l_{i1} \vee l_{i2} \vee \cdots \vee l_{im_i}$$

and all the l_{ij}'s *literals*, i.e. primitive propositions or their negations. The individual conjuncts C_i of a CNF form are often called *clauses*. We usually consider these as sets, since both conjunction and disjunction are associative, commutative and idempotent, so it makes sense to talk of $\bot$ as the empty clause. If C_i consists of one literal, it is called a *unit clause*. Dually, disjunctive normal form (DNF) reverses the role of the 'and's and 'or's. These special forms are analogous to 'fully factorized' and 'fully expanded' in ordinary algebra — think of $(x+1)(x+2)(x+3)$ as CNF and $x^3+6x^2+11x+6$ as DNF. Again by analogy with algebra, we can always translate a formula into CNF by repeatedly rewriting with equivalences like:

$$\begin{array}{rcl}
\neg(\neg p) & \Leftrightarrow & p \\
\neg(p \wedge q) & \Leftrightarrow & \neg p \vee \neg q \\
\neg(p \vee q) & \Leftrightarrow & \neg p \wedge \neg q \\
p \vee (q \wedge r) & \Leftrightarrow & (p \vee q) \wedge (p \vee r) \\
(p \wedge q) \vee r & \Leftrightarrow & (p \vee r) \wedge (q \vee r)
\end{array}$$

However, this in itself can cause the formula to blow up exponentially before we even get to the main algorithm, which is hardly a good start. One can do better by introducing new variables to denote subformulas, and putting the resulting list of equiv-

alences into CNF — so-called *definitional CNF*. It's not hard to see that this preserves satisfiability. For example, we start with the formula:

$$(p \vee (q \wedge \neg r)) \wedge s$$

introduce new variables for subformulas:

$$\begin{array}{l}(p_1 \Leftrightarrow q \wedge \neg r) \wedge \\ (p_2 \Leftrightarrow p \vee p_1) \wedge \\ (p_3 \Leftrightarrow p_2 \wedge s) \wedge \\ p_3\end{array}$$

then transform to CNF:

$$\begin{array}{l}(\neg p_1 \vee q) \wedge (\neg p_1 \vee \neg r) \wedge (p_1 \vee \neg q \vee r) \wedge \\ (\neg p_2 \vee p \vee p_1) \wedge (p_2 \vee \neg p) \wedge (p_2 \vee \neg p_1) \wedge \\ (\neg p_3 \vee p_2) \wedge (\neg p_3 \vee s) \wedge (p_3 \vee \neg p_2 \vee \neg s) \wedge \\ p_3\end{array}$$

The DPLL algorithm is based on the following satisfiability-preserving transformations:

I The 1-literal rule: if a unit clause p appears, remove $\neg p$ from other clauses and remove all clauses including p.

II The affirmative-negative rule: if p occurs *only* negated, or *only* unnegated, delete all clauses involving p.

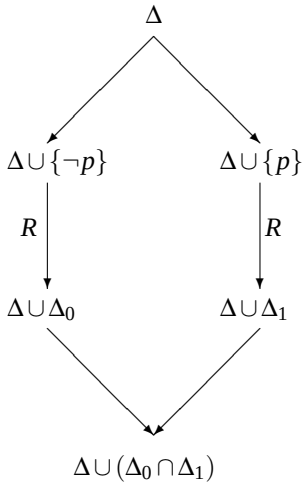
III Case-splitting: consider the two separate problems by adding p and $\neg p$ as new unit clauses.

If you get the empty set of clauses, the formula is satisfiable; if you get an empty *clause*, it is unsatisfiable. Since the first two rules make the problem simpler, one only applies the case-splitting rule when no other progress is possible. In the worst case, many case-splits are necessary and we get exponential behaviour. But in practice it works quite well.

Industrial-strength SAT checkers

The above simple-minded sketch of the DPLL algorithm leaves plenty of room for improvement. The choice of case-splitting variable is often critical, the formulas can be represented in a way that allows for efficient implementation, and the kind of backtracking that arises from case splits can be made more efficient via 'intelligent backjumping' and 'conflict clauses'. Two highly efficient DPLL-based theorem provers are SATO [115] and Chaff [78]. The latter even pays careful attention to low-level issues like cache layout.

Another interesting technique that is used in the tools from Prover Technology (`www.prover.com`), as well as the experimental system Heerhugo [51], is Stålmarck's dilemma rule [102]. This involves using case-splits in a non-nested fashion, accumulating common information from both sides of a case split and feeding it back:



In some cases, this works out much better than the usual DPLL algorithm. For a nice introduction, see [98]. Note that this method is covered by patents [101].

2 Symbolic simulation

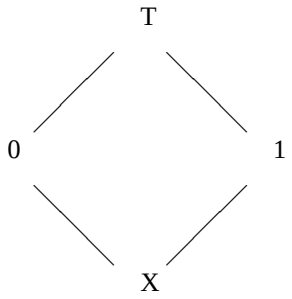
As noted, testing is also widely used in hardware design, and until a decade or so ago was the exclusive way of trying to establish correctness. Usually what is tested is not the circuit itself but some formal model — the whole idea is to flush out errors *before* the expensive step of building the hardware. The usual term for this kind of testing is *simulation*. We'll now consider two refinements of basic simulation, then show how they can profitably be combined.

Ternary simulation

One important optimization of straightforward simulation is *ternary simulation* [16], which uses values drawn from 3-element set: as well as '1' (true) and '0' (false) we have an 'X', denoting an 'unknown' or 'undefined' value. This is *not* meant to imply that a wire in the circuit itself is capable of attaining some third truth-value, but merely reflects the fact that we don't know whether it is 0 or 1. The point of ternary simulation is that many values in the circuit may be irrelevant to some current concern, and we can abstract them away by setting them to 'X' during simulation. (For example, if we are analyzing a small part of a large circuit, e.g. an adder within a complete microprocessor, then only a few bits outside the present module are likely to have any effect on its behaviour and we can set others to X without altering the results of simulation.) We then extend the basic logical operations realized by logic gates to ternary values with this interpretation in mind. For example, we want $0 \wedge X = 0$, since whatever the value on one input, the output will be low if the other input is. Formally, we can present the use of X as an abstraction mapping, for which purpose it's also convenient to add an 'overconstrained' value T, with the intended interpretation:

$$\begin{aligned} T &= \{\} \\ 0 &= \{0\} \\ 1 &= \{1\} \\ X &= \{0,1\} \end{aligned}$$

It's useful to impose an information ordering to give this quaternary lattice:



As expected, the truth tables are monotonic with respect to this ordering:

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
X	X	X	X	X	X
X	0	0	X	X	X
X	1	X	1	1	X
0	X	0	X	1	X
0	0	0	0	1	1
0	1	0	1	1	0
1	X	X	1	X	X
1	0	0	1	0	0
1	1	1	1	1	1

Symbolic simulation

Since we are working in an abstract model, we can easily generalize conventional simulation, which tests on particular combinations of 1s and 0s, to *symbolic simulation* [21], where variables are used for some or all of the inputs, and the outputs are correspondingly symbolic expressions rather than simple truth-values. Testing equivalence then becomes testing of the Boolean expressions. Thus, if we use variables for all inputs, we essentially get back to combinational equivalence checking as we considered before. The main differences are:

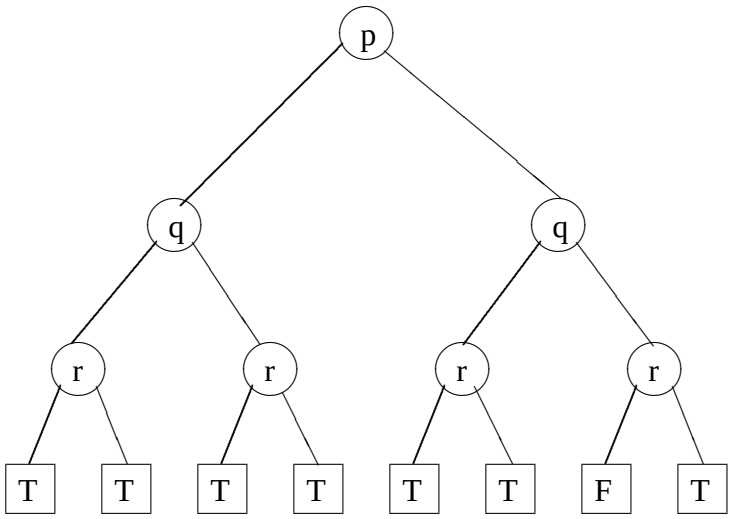
- We may use a more refined circuit model, rather than a simple Boolean switch model

- We explicitly compute the expressions that are the "values" at the internal wires and at the outputs as functions of the input variables, rather than just asserting relations between them.

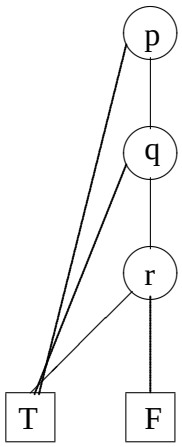
Instead of using general propositional formulas to represent the signals, we can use a canonical representation, where equivalent formulas are represented by the same data structure. Equivalence checking then becomes cheap. The most important such representation is BDDs (binary decision diagrams). Symbolic simulation often does work quite well with BDDs and allows many circuits to be verified exhaustively [14].

BDDs

BDDs are essentially a representation of Boolean functions as decision trees. We can think of them as laying out a truth table but sharing common subcomponents. Consider the truth table for a propositional formula involving primitive propositions $p_1, \ldots, p_n$. Rather than a 2^n-row truth table, we can consider a binary tree indicating how truth assignments for the successive atoms yield a truth value for the whole expression. For example, the function $p \wedge q \Rightarrow q \wedge r$ is represented as follows, where dashed lines indicate the 'false' assignment of a variable, and solid lines the 'true'.



However, there are many common subtrees here, and we can imagine sharing them as much as possible to obtain a directed acyclic graph. Although this degree of saving may be atypical, there is always some potential for sharing, at least in the lowest level of the tree. There are after all "only" 2^{2^k} possible subtrees involving k variables, in particular only two possible leaf nodes 'true' and 'false'.



In general, a representation of a Boolean formula as a binary tree with branch nodes labelled with the primitive atoms and leaf nodes either 'false' or 'true' is called a *binary decision tree*. If all common subtrees are maximally shared to give a directed acyclic graph, it is called a *binary decision diagram* [70, 4]. As part of maximal sharing, we assume that there are no redundant branches, i.e. none where the 'true' and 'false' descendents of a given node are the same subgraph — in such a configuration that subgraph could be replaced by one of its children. If the variable ordering is the same along all branches, we have *reduced ordered binary decision diagrams* (ROBDDs) introduced by Bryant [15]. Nowadays, when people say 'BDD' they normally mean ROBDD, though there have been experiments with non-canonical variants.

Surprisingly many interesting functions have quite a compact BDD representation. However, there are exceptions such as multipliers [15] and the 'hidden weighted bit' function [17]. In these cases it is difficult to apply BDD-based methods directly.

Symbolic trajectory evaluation

A still more refined (and at first sight surprising) approach is to *combine* ternary and symbolic simulation, using Boolean variables to parametrize ternary values. Consider the rather artificial example of verifying that a piece of combinational circuitry with 7 inputs and one output implements a 7-input AND gate. In conventional simulation we would need to check all $2^7 = 128$ input values. In symbolic simulation we would only need to check one case, but that case is a symbolic expression that may be quite large. In ternary simulation, we could verify the circuit only from correct results in 8 explicit cases:

0	X	X	X	X	X	X
X	0	X	X	X	X	X
X	X	0	X	X	X	X
X	X	X	0	X	X	X
X	X	X	X	0	X	X
X	X	X	X	X	0	X
X	X	X	X	X	X	0
1	1	1	1	1	1	1

In a combined approach, we can parametrize these 8 test cases using just 3 Boolean variables (since $2^3 \geq 8$), say p, q and r. For example, we can represent the input wires as

$$a_0 = \begin{cases} 1 & \text{if } p \wedge q \wedge r \\ 0 & \text{if } \neg p \wedge \neg q \wedge \neg r \\ X & \text{otherwise} \end{cases}$$

and

$$a_1 = \begin{cases} 1 & \text{if } p \wedge q \wedge r \\ 0 & \text{if } \neg p \wedge \neg q \wedge r \\ X & \text{otherwise} \end{cases}$$

and so on until:

$$a_6 = \begin{cases} 1 & \text{if } p \wedge q \wedge r \\ 0 & \text{if } p \wedge q \wedge \neg r \\ X & \text{otherwise} \end{cases}$$

Symbolic trajectory evaluation (STE), introduced in [96], uses this kind of parametrization of circuit states in terms of Boolean variables, and a special logic for making correctness assertions, similar to but more restricted than the *temporal logics* we consider in the next lecture. Recently a more general form of STE known as *generalized symbolic trajectory evaluation* (GSTE) has been developed [113]. This can use STE-like methods to verify so-called ω-regular properties, optionally subject to fairness constraints, and so represents a substantial extension of its scope.

STE turns out to be very useful in partially automating the formal verification of some circuits. Part of the appeal is that the use of the ternary model gives a relatively easy way of abstracting out irrelevant detail. STE has been extensively used in formal verification at Intel. For one example, see [80]. For a good introduction to the theory behind STE, see [77].

3 Reachability and model checking

We've mainly been concerned so far with analyzing combinational circuits, although symbolic simulation and STE can be used to track signals over any fixed finite number of clocks. However, it is sometimes of interest to ask questions like 'can a circuit *ever* get into a state where ...' or 'if line r (request) goes high at some point, must line a (acknowledge) eventually do so too?'. Dealing with queries over an unbounded time period like this requires the use of some new techniques.

We can abstract and generalize from the particular features of synchronous sequential digital circuits by considering them as particular cases of a *finite state transition system* or *finite state machine* (FSM). Such a system consists of a finite 'state space' S together with a binary relation $R \subseteq S \times S$ describing the possible transitions, where $R(a,b)$ is true iff it is possible for the system to make a transition from state a to state b in one time-step.[1]

For example, consider a circuit with three latches v_0, v_1 and v_2 that is supposed to implement a modulo-5 counter. The state of the system is described by the values

[1] Sometimes, one also considers a set of possible 'initial' and 'final' states to be part of the state transition system, but we prefer to consider the question of the starting and finishing states separately.

of these latches, so we can choose $S = \{0,1\} \times \{0,1\} \times \{0,1\}$. The corresponding transition relation can be enumerated as follows:

$$\begin{array}{rcl} (0,0,0) & \rightarrow & (0,0,1) \\ (0,0,1) & \rightarrow & (0,1,0) \\ (0,1,0) & \rightarrow & (0,1,1) \\ (0,1,1) & \rightarrow & (1,0,0) \\ (1,0,0) & \rightarrow & (0,0,0) \end{array}$$

The transition systems arising from modelling circuits in this way have the special feature that the transition relation is deterministic, i.e. for each state a there is at most one state b such that $R(a,b)$. However, it's useful to consider state transition systems in general without this restriction, since the same methods can then be applied to a variety of other practically interesting situations, such as the analysis of parallel or nondeterministic hardware, programs or protocols. For example, it is often helpful in analyzing synchronization and mutual exclusion in systems with concurrent and/or interleaved components.

Forward and backward reachability

Many interesting questions are then of the form: if we start in a state $\sigma_0 \in S_0$, can we ever reach a state $\sigma_1 \in S_1$? For example, if we start the counter in state $(0,0,0)$, will it eventually return to the same state? (Fairly obviously, the answer is yes.) In general, we call questions like this *reachability* questions. In order to answer these questions, we can simply construct the reflexive-transitive closure R^* of the transition relation R, and see if it links any pairs of states of interest. In principle, there is no difficulty, since the state space is finite.

Suppose that the set of starting states is S_0; we will consider this the first in an infinite sequence (S_i) of sets of states where S_i is the set of states reachable from S_0 in $\leq i$ transitions. Clearly we can compute S_{i+1} from S_i by the following recursion:

$$S_{i+1} = S_0 \cup \{b \mid \exists a \in S_i.\, R(a,b)\}$$

for a state is reachable in $\leq i+1$ steps if either it is in S_0, i.e. reachable in 0 steps, or it is a possible successor state b to a state a that is reachable in i steps. It is immediate from the intuitive interpretation that $S_i \subseteq S_{i+1}$ for all $i \geq 0$. Now, since each $S_i \subseteq S$, where S is finite, the sets S_i cannot properly increase forever, so we must eventually reach a stage where $S_{i+1} = S_i$ and hence $S_k = S_i$ for all $k \geq i$. It is easy to see that this S_i, which we will write S^*, is then precisely the set of states reachable from S_0 in any finite number of steps.

Thus, we have an algorithm, *forward reachability*, for deciding whether any states in some set $P \subseteq S$ are reachable from S_0. We simply compute S^* and then decide whether $S^* \cap P \neq \emptyset$. In the dual approach, *backward reachability*, we similarly compute the set of states P^* *from which* a state in P is reachable, and then ask whether $S_0 \cap P^* \neq \emptyset$. Once again we can compute the set P^* by iterating an operation until a fixpoint is reached. We can start with $A_0 = P$ and iterate:

$$A_{i+1} = P \cup \{a \mid \exists b \in A_i.\, R(a,b)\}$$

We can simply characterize A_i as the set of states from which a state in P is reachable in $\leq i$ steps. As before, we have $A_i \subseteq A_{i+1}$ and we must eventually reach a fixpoint that is the required set P^*. A formally appealing variant is to use $A_0 = \emptyset$ instead, and the first iteration will give $A_1 = P$ and thereafter we will obtain the same sequence offset by 1 and the same eventual fixpoint P^*. This helps to emphasize that we can consider this a case of finding the least fixpoint of the monotone mapping on sets of states $A \mapsto P \cup \{a \mid \exists b \in A.\, R(a,b)\}$ à la Knaster-Tarski [66, 106].

Is there any reason to prefer forward or backward reachability over the other? Though they will always give the same answer if successfully carried out, one or the other may be much more efficient, depending on the particular system and properties concerned [63]. We will follow the most common approach and focus on backward reachability. This is not because of efficiency concerns, but because, as we will see shortly, it generalizes nicely to analyzing more complicated 'future modalities' than just reachability.

Symbolic state representation

In practice, the kind of fixed-point computation sketched in the previous section can be impractical, because the enumeration of cases in the state set or transition relation is too large. Instead, we can use the techniques we have already established. Instead of representing sets of states and state transitions by explicit enumeration of cases, we can do it *symbolically*, using formulas. Suppose we encode states using Boolean variables $v_1, \ldots, v_n$. Any $T \subseteq S$ can be identified with a Boolean formula $t[v_1, \ldots, v_n]$ that holds precisely when $(v_1, \ldots, v_n) \in T$. Moreover, by introducing n additional 'next state' variables $v'_1, \ldots, v'_n$ we can represent a binary relation A (such as the transition relation R) by a formula $a[v_1, \ldots, v_n, v'_1, \ldots, v'_n]$ that is true iff $A((v_1, \ldots, v_n), (v'_1, \ldots, v'_n))$. For example, we can represent the transition relation for the modulo-5 counter using (among other possibilities) the formula:

$$\begin{array}{l}(v'_0 \Leftrightarrow \neg v_0 \wedge \neg v_2) \wedge \\ (v'_1 \Leftrightarrow \neg(v_0 \Leftrightarrow v_1)) \wedge \\ (v'_2 \Leftrightarrow v_0 \wedge v_1)\end{array}$$

Here the parametrization in terms of Boolean variables just uses one variable for each latch, but this choice, while the most obvious, is not obligatory.

Bounded model checking

One immediate advantage of a symbolic state representation is that we can ask questions about k-step transition sequences without risking a huge blowup in the representation, as might happen in a canonical BDD representation used in symbolic simulation. We can simply duplicate the original formula k times with k sets of the n variables, and can ask any question about the relation between the initial and final states expressed as a Boolean formula $p[v^1_1, \ldots, v^1_n, v^k_1, \ldots, v^k_n]$:

$$\begin{array}{l}r[v^1_1, \ldots, v^1_n, v^2_1, \ldots, v^2_n] \wedge \cdots r[v^{k-1}_1, \ldots, v^{k-1}_n, v^k_1, \ldots, v^k_n] \\ \Rightarrow p[v^1_1, \ldots, v^1_n, v^k_1, \ldots, v^k_n]\end{array}$$

This reduces the problem to tautology-checking. Of course, there is no guarantee that this is feasible in a reasonable amount of time, but experience on many practical

problems with leading-edge tautology checkers has often showed impressive results for this so-called *bounded model checking* [9].

BDD-based reachability

In the case of unbounded queries, the fixpoint computation can easily be performed directly on the symbolic representation. Traditionally, a canonical format such as BDDs has been used [31, 20, 84]. This can immediately cause problems if the BDD representation is infeasibly large, and indeed there have been recent explorations using non-canonical representations [10, 2]. On the other hand, the fact that BDDs are a canonical representation means that if we repeat the iterative steps leading through a sequence S_i, we can immediately recognize when we have reached a fixed point simply by seeing if the BDD for S_{i+1} is identical to that for S_i, whereas in a non-canonical representation, we would need to perform a tautology check at each stage.

Most of the operations used to find the fixpoints are, in the symbolic context, simply logical operations. For example, union ('$\cup$') and intersection ('$\cap$') are simply implemented by conjunction ('$\wedge$') and disjunction ('$\vee$'). However, we also need the operation that maps a set P and a transition relation R to the set $Pre_R(P)$ (or just $Pre(P)$ when R is understood) of states from which there is a 1-step transition into a state in P:

$$Pre(P) = \{a \mid \exists b \in P.\, R(a,b)\}$$

In the symbolic representation, we can characterize this as follows:

$$Pre(p) = \exists v'_1, \ldots, v'_n.\, r[v_1, \ldots, v_n, v'_1, \ldots, v'_n] \wedge p[v'_1, \ldots, v'_n]$$

It is not difficult to implement this procedure on the BDD representation, though it is often a significant efficiency bottleneck.

Temporal logic model checking

There are still many interesting questions about transition systems that we can't answer using just reachability, the request-acknowledge example above being one simple case. But the material so far admits of generalization.

The backward reachability method answered a question 'starting in a state satisfying s, can we reach a state satisfying p?' by computing the set of all states p_* from which p is reachable, and then considering whether $p_* \wedge s = \bot$. If we introduce the notation:

$$EF(p)$$

to mean 'the state p is reachable', or 'there is some state on some path through the relation in which p holds', we can consider p_* as the set of states satisfying the formula $EF(p)$. We can systematically extend propositional logic in this way with further 'temporal operators' to yield a form of so-called *temporal logic*. The most popular logic in practice is *Computation Tree Logic* (CTL), introduced and popularized by Clarke and Emerson [25].

One formal presentation of the semantics of CTL, using separate classes of 'path formulas' and 'state formulas' is the following. (We also need some way of assigning truth-values to primitive propositions based on a state σ, and we will write $[\![\sigma]\!]$ for this mapping applied to σ. In many cases, of course, the properties we are interested in will

be stated precisely in terms of the variables used to encode the states, so a formula is actually its own denotation in the symbolic representation, modulo the transformation from CTL to pure propositional logic.) The valuation functions for an arbitrary formula are then defined as follows for state formulas:

$$\begin{aligned}
sval(R,\sigma)(\bot) &= false \\
sval(R,\sigma)(\top) &= true \\
sval(R,\sigma)(\neg p) &= not(sval(R,\sigma)p) \\
sval(R,\sigma)p &= [\![\sigma]\!](p) \\
&\cdots \\
sval(R,\sigma)(p \Leftrightarrow q) &= (sval(R,\sigma)p = sval(R,\sigma)q) \\
sval(R,\sigma)(Ap) &= \forall\pi.\, Path(R,\sigma)\pi \Rightarrow pval(R,\pi)p \\
sval(R,\sigma)(Ep) &= \exists\pi.\, Path(R,\sigma)\pi \wedge pval(R,\pi)p
\end{aligned}$$

and path formulas:

$$\begin{aligned}
pval(R,\pi)(Fp) &= \exists t.\, sval(R,\pi(t))p \\
pval(R,\pi)(Gp) &= \forall t.\, sval(R,\pi(t))p \\
pval(R,\pi)(p \,\mathtt{U}\, q) &= \exists t.\, (\forall t'.\, t' < t \Rightarrow sval(R,\pi(t'))p) \wedge sval(R,\pi(t))q \\
pval(R,\pi)(Xp) &= sval(R,(t \mapsto \pi(t+1)))p
\end{aligned}$$

Although it was convenient to use path formulas above, the appeal of CTL is exactly that we only really need to consider state formulas, with path formulas just an intermediate concept. We can eliminate the separate class of path formulas by just putting together the path quantifiers and temporal operators in combinations like 'EF' and 'AG'. A typical formula in this logic is the request-acknowledge property:

$$AG(a \Rightarrow EG(r))$$

The process of testing whether a transition system (defining a *model*) satisfies a temporal formula is called *model checking* [25, 88]. The process can be implemented by a very straightforward generalization of the fixpoint methods used in backwards reachability. The marriage of Clarke and Emerson's original explicit-state model-checking algorithms with a BDD-based symbolic representation, due to McMillan [20], gives *symbolic model checking*, and which has led to substantially wider practical applicability. Although at Intel, STE is often more useful, there are situations, e.g. the analysis of bus and cache protocols, where the greater generality of temporal logic model checking seems crucial.

For a detailed discussion of temporal logic model checking, see [26]. The topic is also covered in some books on logic in computer science like [61] and formal verification texts like [69]. CTL is just one example of a temporal logic, and there are innumerable other varieties such as LTL [85], CTL^* [39] and the propositional μ-calculus [68]. LTL stands for *linear temporal logic*, indicating that it takes a different point of view in its semantics, considering all paths and not permitting explicit 'A' and 'E' operators of CTL, but allowing temporal properties to be nested directly without intervening path quantifiers.

4 General theorem proving

It was noted that one of the reasons for the greater success of formal verification in the hardware domain is the fact that more of the verification task can be automated. However, for some more elaborate systems, we cannot rely on tools like equivalence checkers, symbolic simulators or model checkers. One potential obstacle — almost invariably encountered on large industrial designs — is that the system is too large to be verified in a feasible time using such methods. A more fundamental limitation is that some more sophisticated properties cannot be expressed at all in the simple Boolean world. For example, a floating-point *sin* function can hardly have its intended behaviour spelled out in terms of Boolean formulas on the component bits. Instead, we need a more expressive logical system where more or less any mathematical concepts, including real numbers and infinite sets, can be analyzed. This is usually formulated in first-order or higher-order logic, which, as well as allowing propositions to have internal structure, introduces the universal and existential quantifiers:

- The formula $\forall x.\ p$, where x is an (object) variables and p any formula, means intuitively 'p is true for *all* values of x'.
- The analogous formula $\exists x.\ p$ means intuitively 'p is true for *some* value(s) of x', or in other words 'there exists an x such that p'.

I will assume that the basic syntax and definitions of first order logic are known.

Automated theorem proving based on Herbrand's theorem

In contrast to propositional logic, many interesting questions about first order and higher order logic are undecidable even in principle, let alone in practice. Church [24] and Turing [109] showed that even pure logical validity in first order logic is undecidable, introducing in the process many of the basic ideas of computability theory. On the other hand, it is not too hard to see that logical validity is *semidecidable* — this is certainly a direct consequence of completeness theorems for proof systems in first order logic [44], and was arguably implicit in work by Skolem [100]. This means that we can at least program a computer to enumerate all valid first order formulas. One simple approach is based on the following logical principle, due to Skolem and Gödel but usually mis-named "Herbrand's theorem":

> Let $\forall x_1, \ldots, x_n.\ P[x_1, \ldots, x_n]$ be a first order formula with only the indicated universal quantifiers (i.e. the body $P[x_1, \ldots, x_n]$ is quantifier-free). Then the formula is satisfiable iff the infinite set of 'ground instances' $p[t_1^i, \ldots, t_n^i]$ that arise by replacing the variables by arbitrary variable-free terms made up from functions and constants in the original formula is *propositionally* satisfiable.

We can get the original formula into the special form required by some simple normal form transformations, introducing Skolem functions to replace existentially quantified variables. And by compactness for propositional logic, we know that if the infinite set of instances is unsatisfiable, then so will be some finite subset. In principle we can enumerate *all* possible sets, one by one, until we find one that is not propositionally satisfiable. (If the formula is satisfiable, we will never discover it by this means. By undecidability, we know this is unavoidable.) A precise description of this procedure is

tedious, but a simple example may help. Suppose we want to prove that the following is valid. This is often referred to as the 'drinker's principle', because you can think of it as asserting that there is some persom x such that if x drinks, so does everyone.

$$\exists x.\, \forall y.\, D(x) \Rightarrow D(y)$$

We start by negating the formula. To prove that the original is valid, we need to prove that this is unsatisfiable:

$$\neg(\exists x.\, \forall y.\, D(x) \Rightarrow D(y))$$

We then make some transformations to a logical equivalent so that it is in 'prenex form' with all quantifiers at the front.

$$\forall x.\, \exists y.\, D(x) \wedge \neg D(y)$$

We then introduce a Skolem function f for the existentially quantified variable y:

$$\forall x.\, D(x) \wedge \neg D(f(x))$$

We now consider the *Herbrand* universe, the set of all terms built up from constants and functions in the original formula. Since here we have no nullary constants, we need to add one c to get started (this effectively builds in the assumption that all models have a non-empty domain). The Herbrand universe then becomes $\{c, f(c), f(f(c)), f(f(f(c))), \ldots\}$. By Herbrand's theorem, we need to test all sets of ground instances for propositional satisfiability. Let us enumerate them in increasing size. The first one is:

$$D(c) \wedge \neg D(f(c))$$

This is not propositionally unsatisfiable, so we consider the next:

$$(D(c) \wedge \neg D(f(c))) \wedge (D(f(c)) \wedge \neg D(f(f(c))))$$

Now this is propositionally unsatisfiable, so we terminate with success.

Unification-based methods

The above idea [90] led directly some early computer implementations, e.g. by Gilmore [43]. Gilmore tested for propositional satisfiability by transforming the successively larger sets to disjunctive normal form. A more efficient approach is to use the Davis-Putnam algorithm — it was in this context that it was originally introduced [36]. However, as Davis [34] admits in retrospect:

> ... effectively eliminating the truth-functional satisfiability obstacle only uncovered the deeper problem of the combinatorial explosion inherent in unstructured search through the Herbrand universe ...

The next major step forward in theorem proving was a more intelligent means of choosing substitution instances, to pick out the small set of relevant instances instead of blindly trying all possibilities. The first hint of this idea appears in [86], and it was systematically developed by Robinson [92], who gave an effective syntactic procedure called *unification* for deciding on appropriate instantiations to make terms match up correctly.

There are many unification-based theorem proving algorithms. Probably the best-known is *resolution*, in which context Robinson [92] introduced full unification to automated theorem proving. Another important method quite close to resolution and developed independently at about the same time is the inverse method [76, 71]. Other popular algorithms include tableaux [86], model elimination [72, 73] and the connection method [67, 8, 5]. Crudely speaking:

- Tableaux = Gilmore procedure + unification
- Resolution = Davis-Putnam procedure + unification

Tableaux and resolution can be considered as classic representatives of 'top-down' and 'bottom-up' methods respectively. Roughly speaking, in top-down methods one starts from a goal and works backwards, while in bottom-up methods one starts from the assumptions and works forwards. This has significant implications for the very nature of unifiable variables, since in bottom-up methods they are local (implicitly universally quantified) whereas in top-down methods they are global, correlated in different portions of the proof tree. This is probably the most useful way of classifying the various first-order search procedures and has a significant impact on the problems where they perform well.

Decidable problems

Although first order validity is undecidable in general, there are special classes of formulas for which it is decidable, e.g.

- AE formulas, which involve no function symbols and when placed in prenex form have all the universal quantifiers before the existential ones.
- Monadic formulas, involving no function symbols and only monadic (unary) predicate symbols.

The decidability of AE formulas is quite easy to see, because no function symbols are there to start with, and because of the special quantifier nesting, none are introduced in Skolemization. Therefore the Herbrand universe is finite and the enumeration of ground instances cannot go on forever. The decidability of the monadic class can be proved in various ways, e.g. by transforming into AE form by pushing quantifiers inwards ('miniscoping'). Although neither of these classes is particularly useful in practice, it's worth noting that the monadic formulas subsume traditional Aristotelian syllogisms. For example

> If all M are P, and all S are M, then all S are P

can be expressed using monadic predicates as follows:

$$(\forall x.\, M(x) \Rightarrow P(x)) \wedge (\forall x.\, S(x) \Rightarrow M(x)) \Rightarrow (\forall x.\, S(x) \Rightarrow P(x))$$

More interesting in practice are situations where, rather than absolute logical validity, we are interested in whether statements follow from some well-accepted set of mathematical axioms, or are true in some particular model like the real numbers $\mathbb{R}$. In particular, first order formulas built up from equations and inequalities and interpreted over common number systems are often decidable. In the case of the real numbers, one

can use addition and multiplication arbitrarily and it is decidable whether the formula holds in $\mathbb{R}$. This result is originally due to Tarski [105]; for a simpler decision method see [60, 11, 42]. A simple (valid) example is a case of the Cauchy-Schwartz inequality:

$$\forall x_1\ x_2\ y_1\ y_2.\ (x_1 \cdot y_1 + x_2 \cdot y_2)^2 \leq (x_1^2 + x_2^2) \cdot (y_1^2 + y_2^2)$$

Similar results hold for the complex numbers (where of course there is no notion of inequality). In the special case of universally quantified formulas, we can use highly efficient methods like Gröbner bases [18, 32]. If interpreted instead over $\mathbb{N}$ or $\mathbb{Z}$ however, validity of such formulas is not even a semidecidable problem, as a consequence of Gödel's incompleteness theorem [45], and the same applies to $\mathbb{Q}$ [91]. If we restrict ourselves to using just addition (or multiplication by constants, which can be rewritten in terms of addition, e.g. $3 \cdot x = x + x + x$), decidability is regained. This result is due to Presburger [87]; a more efficient decision method is given by Cooper [29]. Surprisingly, one *can* add the exponential function $E(x) = 2^x$, to Presburger arithmetic without losing decidability [97].

Interactive theorem proving

Even though first order validity is semi-decidable, it is seldom practical to solve interesting problems using unification-based approaches to pure logic. Nor is it the case that practical problems often fit conveniently into one of the standard decidable subsets. The best we can hope for in most cases is that the human will have to guide the proof process, but the machine may be able to relieve the tedium by filling in gaps, while always ensuring that no mistakes are made. This kind of application was already envisaged by Wang [110]

> [...] the writer believes that perhaps machines may more quickly become of practical use in mathematical research, not by proving new theorems, but by formalizing and checking outlines of proofs, say, from textbooks to detailed formalizations more rigorous than *Principia* [Mathematica], from technical papers to textbooks, or from abstracts to technical papers.

The first notable interactive provers were the SAM (semi-automated mathematics) series. In 1966, the fifth in the series, SAM V, was used to construct a proof of a hitherto unproven conjecture in lattice theory [19]. This was indubitably a success for the semi-automated approach because the computer automatically proved a result now called "SAM's Lemma" and the mathematician recognized that it easily yielded a proof of Bumcrot's conjecture.

Not long after the SAM project, the AUTOMATH [12, 13] and Mizar [107, 108] proof checking systems appeared, and each of them in its way has been profoundly influential. Although we will refer to these systems as 'interactive', we use this merely as an antonym of 'automatic'. In fact, both AUTOMATH and Mizar were oriented around batch usage. However, the files that they process consist of a *proof*, or a proof sketch, which they *check* the correctness of, rather than a statement for which they attempt to find a proof automatically.

Mizar has been used to proof-check a very large body of mathematics, spanning pure set theory, algebra, analysis, topology, category theory and various unusual applications like mathematical puzzles and computer models. The body of mathematics formally built up in Mizar, known as the 'Mizar Mathematical Library' (MML), seems unrivalled in any other theorem proving system. The 'articles' (proof texts) submitted

to the MML are automatically abstracted into human-readable form and published in the *Journal of Formalized Mathematics*, which is devoted entirely to Mizar formalizations.[2]

LCF — a programmable proof checker

The ideal proof checker should be *programmable*, i.e. users should be able to extend the built-in automation as much as desired. There's no particular difficulty in allowing this. Provided the basic mechanisms of the theorem prover are straightforward and well-documented and the source code is made available, there's no reason why a user shouldn't extend or modify it. However, the difficulty comes if we want to restrict the user to extensions that are logically sound — as presumably we might well wish to, since unsoundness renders questionable the whole idea of machine-checking of supposedly more fallible human proofs. Even fairly simple automated theorem proving programs are often subtler than they appear, and the difficulties of integrating a large body of special proof methods into a powerful interactive system without compromising soundness is not trivial.

One influential solution to this difficulty was introduced in the Edinburgh LCF project led by Robin Milner [50]. Although this was for an obscure 'logic of computable functions' (hence the name LCF), the key idea, as Gordon [48] emphasizes, is equally applicable to more orthodox logics supporting conventional mathematics, and subsequently many programmable proof checkers were designed using the same principles, such as Coq,[3] HOL [49], Isabelle [82] and Nuprl [27].

The key LCF idea is to use a special type (say `thm`) of proven theorems in the implementation language, so that anything of type `thm` must by construction have been *proved* rather than simply asserted. (In practice, the implementation language is usually a version of ML, which was specially designed for this purpose in the LCF project.) This is enforced by making `thm` an *abstract type* whose only constructors correspond to approved inference rules. But the user is given full access to the implementation language and can put the primitive rules together in more complicated ways using arbitrary programming. Because of the abstract type, any result of type `thm`, however it was arrived at, must ultimately have been produced by correct application of the primitive rules. Yet the means for arriving at it may be complex. Most obviously, we can use ML as a kind of 'macro language' to automate common patterns of inference. But much more sophisticated derived rules can be written that, for example, prove formulas of Presburger arithmetic while automatically decomposing to logical primitives. In many theorem-proving tasks, more 'ad hoc' manipulation code can be replaced by code performing inference without significant structural change. In other cases we can use an automated proof procedure, or even an external system like a computer algebra system [58], as an oracle to find a proof that is later checked inside the system. Thus, LCF gives a combination of programmability and logical security that would probably be difficult to assure by other means.

Proof style

One feature of the LCF style is that proofs (being programs) tend to be highly *procedural*, in contrast to the more declarative proofs supported by Mizar — for more on

[2] Available on the Web via `http://www.mizar.org/JFM`.

[3] See the Coq Web page `http://pauillac.inria.fr/coq`.

the contrast see [54]. This can have important disadvantages in terms of readability and maintainability. In particular, it is difficult to understand the formal proof scripts in isolation; they need to be run in the theorem prover to understand what the intermediate states are. Nevertheless as pointed out in [53] it is possible to implement more declarative styles of proof on top of LCF cores. For more recent experiments with Mizar-like declarative proof styles see [104, 111, 114, 112]. Other lectures in this summer school will give an extensive discussion of this topic.

Theorem proving in industry

Theorem provers that have been used in real industrial applications include ACL2 [65], HOL Light [49, 52] and PVS [81]. We noted earlier that formal verification methods can be categorized according to their logical expressiveness and automation. The same kind of balance can be drawn within the general theorem proving section. Although these theorem provers all have undecidable decision problems, it is still possible to provide quite effective partial automation by using a more restricted logic. ACL2 follows this philosophy: it uses a quantifier-free logic analogous to PRA (Primitive Recursive Arithmetic) [47]. HOL and PVS use richer logics with higher-order quantification; PVS's type system is particularly expressive. Nevertheless they attempt to provide some useful automation, and HOL in particular uses the LCF approach to ensure soundness and programmability. This will be emphasized in the application considered below.

5 Floating-point verification

In the present section we describe some work applying HOL Light, an LCF-style prover in the HOL family written by the present author,[4] to some problems in industrial floating-point verification, namely correctness of square root algorithms for the Intel® Itanium® architecture.

Square root algorithms based on `fma`

The centrepiece of the Intel® Itanium® floating-point architecture is the `fma` (floating-point multiply-add or fused multiply-accumulate) family of instructions. Given three floating-point numbers x, y and z, these can compute $x \cdot y \pm z$ as an atomic operation, with the final result rounded as usual according to the IEEE Standard 754 for Binary Floating-Point Arithmetic [62], but without intermediate rounding of the product $x \cdot y$. Of course, one can always obtain the usual addition and multiplication operations as the special cases $x \cdot 1 + y$ and $x \cdot y + 0$.

The `fma` has many applications in typical floating-point codes, where it can often improve accuracy and/or performance. In particular [75] correctly rounded quotients and square roots can be computed by fairly short sequences of `fma`s, obviating the need for dedicated instructions. Besides enabling compilers and assembly language programmers to make special optimizations, deferring these operations to software often yields much higher throughput than with typical hardware implementations. Moreover, the floating-point unit becomes simpler and easier to optimize because minimal hardware need be dedicated to these relatively infrequent operations, and scheduling does not have to cope with their exceptionally high latency.

[4] See `http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`.

Itanium® architecture compilers for high-level languages will typically translate division or square root operations into appropriate sequences of machine instructions. Which sequence is used depends (i) on the required precision and (ii) whether one wishes to minimize latency or maximize throughput. For concreteness, we will focus on a particular algorithm for calculating square roots in double-extended precision (64-bit precision and 15-bit exponent field):

1.	$y_0 = \texttt{frsqrta}(a)$	
2.	$H_0 = \frac{1}{2} y_0$	$S_0 = a y_0$
3.	$d_0 = \frac{1}{2} - S_0 H_0$	
4.	$H_1 = H_0 + d_0 H_0$	$S_1 = S_0 + d_0 S_0$
5.	$d_1 = \frac{1}{2} - S_1 H_1$	
6.	$H_2 = H_1 + d_1 H_1$	$S_2 = S_1 + d_1 S_1$
7.	$d_2 = \frac{1}{2} - S_2 H_2$	$e_2 = a - S_2 S_2$
8.	$H_3 = H_2 + d_2 H_2$	$S_3 = S_2 + e_2 H_2$
9.	$e_3 = a - S_3 S_3$	
10.	$S = S_3 + e_3 H_3$	

All operations but the last are done using the register floating-point format with rounding to nearest and with all exceptions disabled. (This format provides the same 64-bit precision as the target format but has a greater exponent range, allowing us to avoid intermediate overflow or underflow.) The final operation is done in double-extended precision using whatever rounding mode is currently selected by the user.

This algorithm is a non-trivial example in two senses. Since it is designed for the maximum precision supported in hardware (64 bits), greater precision cannot be exploited in intermediate calculations and so a very careful analysis is necessary to ensure correct rounding. Moreover, it is hardly feasible to test such an algorithm exhaustively, even if an accurate and fast reference were available, since there are about 2^{80} possible inputs. (By contrast, one could certainly verify single-precision and conceivably verify double precision by exhaustive or quasi-exhaustive methods.)

Algorithm verification

It's useful to divide the algorithm into three parts, and our discussion of the correctness proof will follow this separation:

1 Form[5] an initial approximation $y_0 = \frac{1}{\sqrt{a}}(1+\varepsilon)$ with $|\varepsilon| \leq 2^{-8.8}$.

2–8 Convert this to approximations $H_0 \approx \frac{1}{2\sqrt{a}}$ and $S_0 \approx \sqrt{a}$, then successively refine these to much better approximations H_3 and S_3 using Goldschmidt iteration [46] (a Newton-Raphson variant).

9–10 Use these accurate approximations to produce the square root S correctly rounded according to the current rounding mode, setting IEEE flags or triggering exceptions as appropriate.

[5] Using `frsqrta`, the only Itanium® instruction specially intended to support square root. In the present discussion we abstract somewhat from the actual machine instruction, and ignore exceptional cases like $a = 0$ where it takes special action.

Initial approximation

The `frsrta` instruction makes a number of initial checks for special cases that are dealt with separately, and if necessary normalizes the input number. It then uses a simple table lookup to provide the approximation. The algorithm and table used are precisely specified in the Itanium® instruction set architecture. The formal verification is essentially some routine algebraic manipulations for exponent scaling, then a 256-way case split followed by numerical calculation. The following HOL theorem concerns the correctness of the core table lookup:

```
|- normal a ∧ &0 <= Val a
   ⇒ abs(Val(frsqrta a) / inv(sqrt(Val a)) - &1)
            < &303 / &138050
```

Refinement

Each `fma` operation will incur a rounding error, but we can easily find a mathematically convenient (though by no means optimally sharp) bound for the relative error induced by rounding. The key principle is the '$1+e$' property, which states that the rounded result involves only a small relative perturbation to the exact result. In HOL the formal statement is as follows:

```
|- ¬(losing fmt rc x) ∧ ¬(precision fmt = 0)
   ⇒ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
           (round fmt rc x = x * (&1 + e))
```

The bound on e depends on the precision of the floating-point format and the rounding mode; for round-to-nearest mode, `mu rc` is $1/2$. The theorem has two side conditions, one being a nontriviality hypothesis, and the other an assertion that the value x does not *lose precision*. We will not show the formal definition [55] here, since it is rather complicated. However, a simple and usually adequate sufficient condition is that the exact result lies in the normal range (or is zero).

Actually applying this theorem, and then bounding the various error terms, would be quite tedious if done by hand. We have programmed some special derived rules in HOL to help us. First, these automatically bound absolute magnitudes of quantities, essentially by using the triangle rule $|x+y| \leq |x|+|y|$. This usually allows us to show that no overflow occurs. However, to apply the $1+e$ theorem, we also need to exclude underflow, and so must establish *minimum* (nonzero) absolute magnitudes. This is also largely done automatically by HOL, repeatedly using theorems for the minimum nonzero magnitude that can result from an individual operation. For example, if $2^e \leq |a|$, then either $a+b\cdot c$ is exactly zero or $2^{e-2p} \leq |a+b\cdot c|$ where p is the precision of the floating-point format containing a, b and c.

It's now quite easy with a combination of automatic error bounding and some manual algebraic rearrangement to obtain quite good relative error bounds for the main computed quantities. In fact, in the early iterations, the rounding errors incurred are insignificant in comparison with the approximation errors in the H_i and S_i. Thus, the relative errors in these quantities are roughly in step. If we write

$$H_i \approx \frac{1}{2\sqrt{a}}(1+\varepsilon_i) \qquad S_i \approx \sqrt{a}(1+\varepsilon_i)$$

then

$$d_i \approx \frac{1}{2} - S_i H_i = \frac{1}{2} - \frac{1}{2}(1+\varepsilon_i)^2 = -(\varepsilon_i + \varepsilon_i^2/2)$$

Consequently, correcting the current approximations in the manner indicated will approximately square the relative error, e.g.

$$S_{i+1} \approx S_i + d_i S_i = S_i(1+d_i) \approx \sqrt{a}(1+\varepsilon_i)(1-\varepsilon_i-\varepsilon_i^2/2) = \sqrt{a}(1-\frac{3}{2}\varepsilon_i^2)$$

Towards the end, the rounding errors in S_i and H_i become more significantly decoupled and for the penultimate iteration we use a slightly different refinement for S_3.

$$e_2 \approx a - S_2 S_2 = a - (\sqrt{a}(1+\varepsilon_2)^2) \approx -2a\varepsilon_2$$

and so:

$$S_2 + e_2 H_2 \approx \sqrt{a}(1+\varepsilon_2) - (2a\varepsilon_2)(\frac{1}{2\sqrt{a}}(1+\varepsilon_2')) \approx \sqrt{a}(1-\varepsilon_2\varepsilon_2')$$

Thus, $S_2 + e_2 H_2$ will be quite an accurate square root approximation. In fact the HOL proof yields $S_2 + e_2 H_2 = \sqrt{a}(1+\varepsilon)$ with $|\varepsilon| \leq 5579/2^{79} \approx 2^{-66.5}$.

The above sketch elides what in the HOL proofs is a detailed bound on the rounding error. However this only really becomes significant when S_3 is rounded; this may in itself contribute a relative error of order 2^{-64}, significantly more than the error before rounding. Nevertheless it is important to note that if $\sqrt{a}$ happens to be an exact floating-point number (e.g. $\sqrt{1.5625} = 1.25$), S_3 will be that number. This is a consequence of the fact that the error in $S_2 + e_2 H_2$ is less than half the distance between surrounding floating-point numbers.

Correct rounding

The final two steps of the algorithm simply repeat the previous iteration for S_3 and the basic error analysis is the same. The difficulty is in passing from a relative error before rounding to correct rounding afterwards. Again we consider the final rounding separately, so S is the result of rounding the exact value $S^* = S_3 + e_3 H_3$. The error analysis indicates that $S^* = \sqrt{a}(1+\varepsilon)$ for some $|\varepsilon| \leq 25219/2^{140} \approx 2^{-125.37}$. The final result S will, by the basic property of the `fma` operation, be the result of rounding S^* in whatever the chosen rounding mode may be. The *desired* result would be the result of rounding exactly $\sqrt{a}$ in the same way. How can we be sure these are the same?

First we can dispose of some special cases. We noted earlier that if $\sqrt{a}$ is already exactly a floating-point number, then S_3 will already be that number. In this case we will have $e_3 = 0$ and so $S^* = S_3$. Whatever the rounding mode, rounding a number already in the format concerned will give that number itself:

```
|- a IN iformat fmt ⇒ (round fmt rc a = a)
```

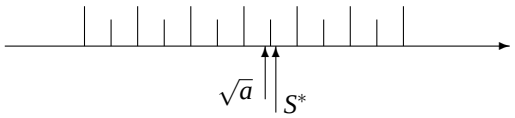
so the result will be correct. Moreover, the earlier observation extends to show that if $\sqrt{a}$ is fairly close (in a precise sense) to a floating-point number, then S_3 will be that number. It is then quite straightforward to see that the overall algorithm will be accurate without great subtlety: we just need the fact that e_3 has the right sign and roughly the correct magnitude, so S^* will never misround in directed rounding modes.

Thus, we can also deal immediately with what would otherwise be difficult cases for the directed rounding modes, and concentrate our efforts on rounding to nearest.

On general grounds we note that $\sqrt{a}$ *cannot* be exactly the mid-point between two floating-point numbers. This is not hard to see, since the square root of a number in a given format cannot denormalize in that format, and a non-denormal midpoint has $p+1$ significant digits, so its square must have more than p.[6]

```
|- &0 <= a ∧ a IN iformat fmt ∧ b IN midpoints fmt
   ⇒ ¬(sqrt a = b)
```

This is a useful observation. We'll never be in the tricky case where there are two equally close floating-point numbers (resolved by the 'round to even' rule.) So in round-to-nearest, S^* and $\sqrt{a}$ could only round in different ways if there were a midpoint between them, for only then could the closest floating-point numbers to them differ. For example in the following diagram where large lines indicate floating-point numbers and smaller ones represent midpoints, $\sqrt{a}$ would round 'down' while S^* would round 'up':[7]



Although analyzing this condition combinatorially would be complicated, there is a much simpler sufficient condition. One can easily see that it would suffice to show that for any midpoint m:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* couldn't lie on opposite sides of m. Here is the formal theorem in HOL:

```
|- ¬(precision fmt = 0) ∧
   (∀m. m IN midpoints fmt ⇒ abs(x - y) < abs(x - m))
   ⇒ (round fmt Nearest x = round fmt Nearest y)
```

One can arrive at an 'exclusion zone' theorem giving the minimum possible $|\sqrt{a} - m|$. However, this can be quite small, about $2^{-(2p+3)}$ relative to $\sqrt{a}$, where p is the precision. For example, in our context with $p = 64$, consider the square root of the next floating-point number below 1, whose mantissa consists entirely of 1s. Its square root is about 2^{-131} from a midpoint:

$$\sqrt{1 - 2^{-64}} \approx (1 - 2^{65}) - 2^{-131}$$

Therefore, our relative error in S^* of about $2^{-125.37}$ is far from adequate to justify perfect rounding based on the simple 'exclusion zone' theorem, for which we need something of order 2^{-131}. However, our relative error bounds are far from sharp, and it seems quite plausible that the algorithm does nevertheless work correctly. What can we do?

[6] An analogous result holds for quotients but here the denormal case must be dealt with specially. For example $2^{E_{min}} \times 0.111\cdots111/2$ is exactly a midpoint.

[7] Similarly, in the other rounding modes, misrounding could only occur if $\sqrt{a}$ and S^* are separated by a floating-point number. However as we have noted one can deal with those cases more directly.

One solution is to utilize more refined theorems [74], but this is complicated and may still fail to justify several algorithms that are intuitively believed to work correctly. An ingenious alternative developed by Cornea [30] is to observe that there are relatively few cases like $0.111\cdots1111$ whose square roots come close enough to render the exclusion zone theorem inapplicable, and these can be isolated by fairly straightforward number-theoretic methods. We can therefore:

- Isolate the special cases $a_1, \ldots, a_n$ that have square roots within the critical distance of a midpoint.
- Conclude from the simple exclusion zone theorem that the algorithm will give correct results except possibly for $a_1, \ldots, a_n$.
- Explicitly show that the algorithm is correct for the $a_1, \ldots, a_n$, (effectively by running it on those inputs).

This two-part approach is perhaps a little unusual, but not unknown even in pure mathematics.[8] For example, consider "Bertrand's Conjecture" (first proved by Chebyshev), stating that for any positive integer n there is a prime p with $n \leq p \leq 2n$. The most popular proof [40], involves assuming $n > 4000$ for the main proof and separately checking the assertion for $n \leq 4000$.[9]

By some straightforward mathematics described in [30] and formalized in HOL without difficulty, one can show that the difficult cases for square roots have mantissas m, considered as p-bit integers, such that one of the following diophantine equations has a solution k for some integer $|d| \leq D$, where D is roughly the factor by which the guaranteed relative error is excessive:

$$2^{p+2}m = k^2 + d \qquad 2^{p+1}m = k^2 + d$$

We consider the equations separately for each chosen $|d| \leq D$. For example, we might be interested in whether $2^{p+1}m = k^2 - 7$ has a solution. If so, the possible value(s) of m are added to the set of difficult cases. It's quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

$$\vdash (2^{p+1}m = k^2 + d) \Rightarrow (m = n_1) \vee \ldots \vee (m = n_i)$$

The procedure simply uses even-odd reasoning and recursion on the power of two (effectively so-called 'Hensel lifting'). For example, if

$$2^{25}m = k^2 - 7$$

then we know k must be odd; we can write $k = 2k' + 1$ and deduce:

$$2^{24}m = 2k'^2 + 2k' - 3$$

By more even/odd reasoning, this has no solutions. In general, we recurse down to an equation that is trivially unsatisfiable, as here, or immediately solvable. One equation can split into two, but never more. For example, we have a formally proved

[8] A more extreme case is the 4-color theorem, whose proof relies on extensive (computer-assisted) checking of special cases [6].

[9] An 'optimized' way of checking, referred to in [3] as "Landau's trick", is to verify that 3, 5, 7, 13, 23, 43, 83, 163, 317, 631, 1259, 2503 and 4001 are all prime and each is less than twice its predecessor.

HOL theorem asserting that for any double-extended number a,[10] rounding $\sqrt{a}$ and $\sqrt{a}(1+\varepsilon)$ to double-extended precision using any of the four IEEE rounding modes will give the same results provided $|\varepsilon| < 31/2^{131}$, with the possible exceptions of $2^{2e}m$ for:

$$\begin{aligned} m \in \{\ & 10074057467468575321, 10376293541461622781, \\ & 10376293541461622787, 11307741603771905196, \\ & 13812780109330227882, 14928119304823191698, \\ & 16640932189858196938, 18446744073709551611, \\ & 18446744073709551612, 18446744073709551613, \\ & 18446744073709551614, 18446744073709551615\} \end{aligned}$$

and $2^{2e+1}m$ for

$$\begin{aligned} m \in \{\ & 9223372036854775809, 9223372036854775811, \\ & 11168682418930654643\} \end{aligned}$$

Note that while some of these numbers are obvious special cases like $2^{64}-1$, the "pattern" in others is only apparent from the kind of mathematical analysis we have undertaken here. They aren't likely to be exercised by random testing, or testing of plausible special cases.[11]

Checking formally that the algorithm works on the special cases can also be automated, by applying theorems on the uniqueness of rounding to the concrete numbers computed. (For a formal proof, it is not sufficient to separately test the implemented algorithm, since such a result has no formal status.) In order to avoid trying all possible even or odd exponents for the various significands, we exploit some results on invariance of the rounding and arithmetic involved in the algorithm under systematic scaling by 2^{2k}, doing a simple form of symbolic simulation by formal proof.

Flag settings

Correctness according to the IEEE Standard 754 not only requires the correctly rounded result, but the correct setting of flags or triggering of exceptions for conditions like overflow, underflow and inexactness. Actually, almost all these properties follow directly from the arguments leading to perfect rounding. For example, the mere fact that two real numbers round equivalently *in all rounding modes* implies that one is exact iff the other is:

```
|- ¬(precision fmt = 0) ∧
   (∀rc. round fmt rc x = round fmt rc y)
   ⇒ ∀rc. (round fmt rc x = x) = (round fmt rc y = y)
```

The correctness of other flag settings follows in the same sort of way, with underflow only slightly more complicated [55].

[10] Note that there is more subtlety required when using such a result in a mixed-precision environment. For example, to obtain a single-precision result for a double-precision input, an algorithm that suffices for single-precision inputs may not be adequate even though the final precision is the same.

[11] On the other hand, we can well consider the mathematical analysis as a *source* of good test cases.

Conclusions and related work

What do we gain from developing these proofs formally in a theorem prover, compared with a detailed hand proof? We see two main benefits: reliability and re-usability.

Proofs of this nature, large parts of which involve intricate but routine error bounding and the exhaustive solution of Diophantine equations, are very tedious and error-prone to do by hand. In practice, one would do better to use *some* kind of machine assistance, such as *ad hoc* programs to solve the Diophantine equations and check the special cases so derived. Although this can be helpful, it can also create new dangers of incorrectly implemented helper programs and transcription errors when passing results between 'hand' and 'machine' portions of the proof. By contrast, we perform all steps of the proof in a painstakingly foundational system, and can be quite confident that no errors have been introduced. The proof proceeds according to strict logical deduction, all the way from the underlying pure mathematics up to the symbolic "execution" of the algorithm in special cases.

Although we have only discussed one particular example, many algorithms with a similar format have been developed for use in systems based on the Itanium® architecture. One of the benefits of implementing division and square root in software is that different algorithms can be substituted depending on the detailed accuracy and performance requirements of the application. Not only are different (faster) algorithms provided for IEEE single and double precision operations, but algorithms often have two versions, one optimized for minimum latency and one for maximal throughput. These algorithms are all quite similar in structure and large parts of the correctness proofs use the same ideas. By performing these proofs in a programmable theorem prover like HOL, we are able to achieve high re-use of results, just tweaking a few details each time. Often, we can produce a complete formal proof of a new algorithm in just a day. For a even more rigidly stereotyped class of algorithms, one could quite practically implement a totally automatic verification rule in HOL.

Underlying these advantages are three essential theorem prover features: soundness, programmability and executability. HOL scores highly on these points. It is implemented in a highly foundational style and does not rely on the correctness of very complex code. It is freely programmable, since it is embedded in a full programming language. In particular, one can program it to perform various kinds of computation and symbolic execution by proof. The main disadvantage is that proofs can sometimes take a long time to run, precisely because they *always* decompose to low-level primitives. This applies with particular force to some kinds of symbolic execution, where instead of simply accepting an equivalence like $2^{94}+3 = 13\cdot 19\cdot 681943\cdot 7941336391\cdot 14807473717$ based, say, on the results of a multiprecision arithmetic package, a detailed formal proof is constructed under the surface. To some extent, this sacrifice of efficiency is a conscious choice when we decide to adopt a highly foundational system, but it might be worth weakening this ideology at least to include concrete arithmetic as an efficient primitive operation.

This is by no means the only work in this area. On the contrary, floating-point verification is regarded as one of the success stories of formal verification. For related work on square root algorithms in commercial systems see [93, 80, 94]. For similar verifications of division algorithms and transcendental functions by the present author, see [57, 56], while [79] and [74] give a detailed discussion of some relevant floating-point algorithms. When verifying transcendental functions, one needs a fair amount of pure mathematics formalized just to get started. So, in a common intellectual pattern, it is entirely possible that Mizar-like formalizations of mathematics undertaken for purely

intellectual reasons might in the end turn out to be useful in practical applications.

References

[1] M. Aagaard and M. Leeser. Verifying a logic synthesis tool in Nuprl: A case study in software verification. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification: Proceedings of the Fourth International Workshop, CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 69–81, Montreal, Canada, 1994. Springer Verlag.

[2] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[3] M. Aigner and G. M. Ziegler. *Proofs from The Book*. Springer-Verlag, 2nd edition, 2001.

[4] S. B. Akers. Binary decision diagrams. *ACM Transactions on Computers*, C-27:509–516, 1978.

[5] P. B. Andrews. Theorem proving by matings. *IEEE transactions on computers*, 25:801–807, 1976.

[6] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82:711–712, 1976.

[7] Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors. *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, Nice, France, 1999. Springer-Verlag.

[8] W. Bibel and J. Schreiber. Proof search in a Gentzen-like system of first order logic. In E. Gelenbe and D. Potier, editors, *Proceedings of the International Computing Symposium*, pages 205–212. North-Holland, 1975.

[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[10] P. Bjesse. Symbolic model checking with sets of states represented as formulas. Technical Report SC-1999-100, Department of Computer Science, Chalmers University of Technology, 1999.

[11] J. Bochnak, M. Coste, and M.-F. Roy. *Real Algebraic Geometry*, volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, 1998.

[12] N. G. d. Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[13] N. G. d. Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[14] R. E. Bryant. Symbolic verification of MOS circuits. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 419–438. Computer Science Press, 1985.

[15] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[16] R. E. Bryant. A method for hardware verification based on logic simulation. *Journal of the ACM*, 38:299–328, 1991.

[17] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, C-40:205–213, 1991.

[18] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Mathematisches Institut der Universität Innsbruck, 1965.

[19] R. Bumcrot. On lattice complements. *Proceedings of the Glasgow Mathematical Association*, 7:22–23, 1965.

[20] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[21] W. C. Carter, W. H. Joyner, and D. Brand. Symbolic simulation for correct machine design. In *Proceedings of the 16th ACM/IEEE Design Automation Conference*, pages 280–286. IEEE Computer Society Press, 1979.

[22] B. F. Caviness and J. R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and monographs in symbolic computation. Springer-Verlag, 1998.

[23] C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. *Journal of Automated Reasoning*, 23:265–298, 1999.

[24] A. Church. An unsolvable problem of elementary number-theory. *American Journal of Mathematics*, 58:345–363, 1936.

[25] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, 1981. Springer-Verlag.

[26] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[27] R. Constable. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall, 1986.

[28] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[29] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Melzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. Elsevier, 1972.

[30] M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1–11, 1998. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.

[31] O. Coudert, C. Berthet, and J.-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, 1989.

[32] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag, 1992.

[33] M. Davis, editor. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Raven Press, NY, 1965.

[34] M. Davis. The prehistory and early history of automated deduction. In Siekmann and Wrightson [99], pages 1–28.

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

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

[37] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22:271–280, 1979.

[38] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[39] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33:151–178, 1986.

[40] P. Erdös. Beweis eines Satzes von Tschebyshev. *Acta Scientiarum Mathematicarum (Szeged)*, 5:194–198, 1930.

[41] J. H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31:1048–1063, 1988.

[42] L. Gårding. *Some Points of Analysis and Their History*, volume 11 of *University Lecture Series*. American Mathematical Society / Higher Education Press, 1997.

[43] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM Journal of research and development*, 4:28–35, 1960.

[44] K. Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. English translation 'The completeness of the axioms of the functional calculus of logic' in [59], pp. 582–591.

[45] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English translation, 'On Formally Undecidable Propositions of Principia Mathematica and Related Systems, I', in [59], pp. 592–618 or [33], pp. 4–38.

[46] R. E. Goldschmidt. Applications of division by convergence. Master's thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass., 1964.

[47] R. L. Goodstein. *Recursive Number Theory*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1957.

[48] M. J. C. Gordon. Representing a logic in the LCF metalanguage. In D. Néel, editor, *Tools and notions for program construction: an advanced course*, pages 163–185. Cambridge University Press, 1982.

[49] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[50] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[51] J. F. Groote. The propositional formula checker Heerhugo. *Journal of Automated Reasoning*, 24:101–125, 2000.

[52] J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[53] J. Harrison. A Mizar mode for HOL. In J. v. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.

[54] J. Harrison. Proof style. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172, Aussois, France, 1996. Springer-Verlag.

[55] J. Harrison. A machine-checked theory of floating point arithmetic. In Bertot et al. [7], pages 113–130.

[56] J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.

[57] J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[58] J. Harrison and L. Théry. A sceptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[59] J. v. Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931*. Harvard University Press, 1967.

[60] L. Hörmander. *The Analysis of Linear Partial Differential Operators II*, volume 257 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1983.

[61] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 1999.

[62] IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.

[63] H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *Proceedings of tte IEEE/ACM conference on Computer Aided Design (ICCAD '96)*, pages 82–87. Association for Computing Machinery, 1996.

[64] J. J. Joyce and C. Seger. The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. In J. J. Joyce and C. Seger, editors, *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, volume 780 of *Lecture Notes in Computer Science*, pages 185–198, UBC, Vancouver, Canada, 1993. Springer-Verlag.

[65] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[66] B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134, 1927. Volume published in 1928.

[67] R. Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22:572–595, 1975.

[68] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[69] T. Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

[70] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[71] V. Lifschitz. *Mechanical Theorem Proving in the USSR: the Leningrad School*. Monograph Series on Soviet Union. Delphic Associates, 7700 Leesburg Pike, #250, Falls Church, VA 22043. Phone: (703) 556-0278, 1986. See also 'What is the inverse method?' in the Journal of Automated Reasoning, vol. 5, pp. 1–23, 1989.

[72] D. W. Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM*, 15:236–251, 1968.

[73] D. W. Loveland. *Automated theorem proving: a logical basis*. North-Holland, 1978.

[74] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall, 2000.

[75] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34:111–119, 1990.

[76] S. J. Maslov. An inverse method of establishing deducibility in classical predicate calculus. *Doklady Akademii Nauk*, 159:17–20, 1964.

[77] T. Melham and A. Darbari. Symbolic trajectory evaluation in a nutshell. 2002.

[78] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM Press, 2001.

[79] J.-M. Muller. *Elementary functions: Algorithms and Implementation*. Birkhäuser, 1997.

[80] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.

[81] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[82] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.

[83] D. A. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.

[84] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In *Proceedings of the DIMACS workshop on Computer Aided Verification*, pages 293–320. DIMACS (technical report 90-31), 1990.

[85] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–67, 1977.

[86] D. Prawitz, H. Prawitz, and N. Voghera. A mechanical proof procedure and its realization in an electronic computer. *Journal of the ACM*, 7:102–128, 1960.

[87] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pages 92–101, 395. Warsaw, 1930. Annotated English version by [103].

[88] J. P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 195–220. Springer-Verlag, 1982.

[89] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model-checking with automated proof-checking. In P. Wolper, editor, *Computer-Aided Verification: CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, 1995. Springer-Verlag.

[90] A. Robinson. Proving a theorem (as done by man, logician, or machine). In *Summaries of Talks Presented at the Summer Institute for Symbolic Logic*, 1957. Second edition published by the Institute for Defense Analysis, 1960. Reprinted in [99], pp. 74–76.

[91] J. Robinson. Definability and decision problems in arithmetic. *Journal of Symbolic Logic*, 14:98–114, 1949. Author's PhD thesis.

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

[93] D. Rusinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. Available on the Web via `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.

[94] J. Sawada and R. Gamboa. Mechanical verification of a square root algorithms using taylor's theorem. In M. Aagaard and J. O'Leary, editors, *Formal Methods in Computer-Aided Design: Fourth International Conference FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[95] C. Seger and J. J. Joyce. A two-level formal verification methodology using HOL and COSMOS. Technical Report 91-10, Department of Computer Science, University of British Columbia, 2366 Main Mall, University of British Columbia, Vancouver, B.C, Canada V6T 1Z4, 1991.

[96] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, 1995.

[97] A. L. Semënov. Logical theories of one-place functions on the set of natural numbers. *Mathematics of the USSR Izvestiya*, 22:587–618, 1984.

[98] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. J. Windley, editors, *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99. Springer-Verlag, 1998.

[99] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning — Classical Papers on Computational Logic, Vol. I (1957-1966)*. Springer-Verlag, 1983.

[100] T. Skolem. Einige Bemerkungen zur axiomatischen Begründung der Mengenlehre. In *Matematikerkongress i Helsingfors den 4–7 Juli 1922, Den femte skandinaviska matematikerkongressen, Redogörelse*. Akademiska Bokhandeln, Helsinki, 1922. English translation "Some remarks on axiomatized set theory" in [59], pp. 290–301.

[101] G. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula. United States Patent number 5,276,897; see also Swedish Patent 467 076, 1994.

[102] G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, pages 31–36, Gatwick, UK, 1990. Pergamon Press.

[103] R. Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical Report CORNELLCS:TR84-639, Cornell University Computer Science Department, 1984.

[104] D. Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997.

[105] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. Previous version published as a technical report by the RAND Corporation, 1948; prepared for publication by J. C. C. McKinsey. Reprinted in [22], pp. 24–84.

[106] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[107] A. Trybulec. The Mizar-QC/6000 logic information language. *ALLC Bulletin (Association for Literary and Linguistic Computing)*, 6:136–140, 1978.

[108] A. Trybulec and H. A. Blair. Computer aided reasoning. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 406–412, Brooklyn, 1985. Springer-Verlag.

[109] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (2)*, 42:230–265, 1936.

[110] H. Wang. Toward mechanical mathematics. *IBM Journal of research and development*, 4:2–22, 1960.

[111] M. Wenzel. Isar - a generic intepretive approach to readable formal proof documents. In Bertot et al. [7], pages 167–183.

[112] F. Wiedijk. Mizar light for HOL Light. In R. J. Boulton and P. B. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 378–394. Springer-Verlag, 2001. Online at `http://link.springer.de/link/service/series/0558/tocs/t2152.htm`.

[113] J. Yang. A theory for generalized symbolic trajectory evaluation. In *Proceedings of the 2000 Symposium on Symbolic Trajectory Evaluation*, Chicago, 2000. Available via `http://www.intel.com/research/scl/stesympsite.htm`.

[114] V. Zammit. On the implementation of an extensible declarative proof language. In Bertot et al. [7], pages 185–202.

[115] H. Zhang. SATO: an efficient propositional prover. In W. McCune, editor, *Automated Deduction — CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275, Townsville, Australia, 1997. Springer-Verlag.