AUTOMATED THEOREM PROVING: AN OVERVIEW

Talal H. Maghrabi*

Information and Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia

الخلاصة :

تُستخدم نظم إثبات النظريات آلياً في المساعدة في إثبات النظريات الرياضية وغير النظرية . ويلاحظ سرعة التقدم والتطور في مجالات البحوث المتشعبة المتعلقة بإثبات النظريات آلياً .

يقدم هذا البحث دراسة شاملة لهذا المجال المهم ، حيث يَستعرض الخلفية الرياضية اللازمة لفهم الموضوع ، كما يقدّم عرضاً تاريخيا لمراحل تطور البحوث في هذا المجال . ويتطرق البحث إلى المصطلحات الأساسية والرموز المستخدمة ، ويقُدم وصفاً للعناصر الأساسية التي يجب أن تتوفر في نظم إثبات النظريات . ويغطي البحث أيضاً نبذه البحوث الحالية الجارية في مجال إثبات النظريات آلياً مع إعطاء وصف مختصر لبعض النظم المتوفرة حالياً .

ABSTRACT

Automated Theorem Provers are computer programs written to prove, or help in proving, mathematical and non-mathematical theorems. Automated Theorem Proving (ATP) is a rapidly advancing field and contains many potential research areas. This paper is an overview of this important field. It starts by giving the needed mathematical background followed by an overview. The overview includes historical background, basic terminology and notations and a description of the major components of a typical theorem prover. The paper also outlines the current state of research in ATP and a brief description of some of the existing theorem provers.

*Address for correspondence:
KFUPM Box 806
King Fahd University of Petroleum & Minerals
Dhahran 31261
Saudi Arabia

Talal H. Maghrabi

AUTOMATED THEOREM PROVING: AN OVERVIEW

1. INTRODUCTION

Automated theorem provers are computer programs written to prove, or help in proving, mathematical and non-mathematical theorems. Automated Theorem Proving (ATP) first emerged in the late 1950s, when mathematicians became interested in automating mathematical proving techniques. Since that time ATP has evolved rapidly and many changes have taken place. Currently, ATP is considered to be a fast growing field of computer science. It has many application areas such as *program verification*, *program synthesis*, *hardware verification*, and *mathematics*. Many general and special-purpose theorem provers have been developed in the last few years, and many more are expected to be developed in the near future. In this paper we present an overview of this important field and try to enlighten the reader about its history and future. Section 2 gives some mathematical background which mainly concentrates on First Order Logic (FOL) since it is the basic foundation of ATP. Section 3 summarizes the history of ATP and gives basic terminology and notations. It also describes the major components of a typical theorem prover. Section 4 outlines the current status of the field and its research areas. It also describes some of the well-known theorem provers. Finally, Section 5 provides an example of a problem from the field of mathematics that is solved using the OTTER theorem prover.

2. MATHEMATICAL BACKGROUND

In this section we give a brief introduction to FOL, which is the basic foundation for automated theorem proving. First we define what logic is, and then we describe the basic elements of FOL.

2.1. Deductive Systems

A *logic* (or *a deductive system*) consists of a *language*, a *proof theory*, and a *model theory*. The language consists of: a countable set S of symbols called the *alphabet*, and a set of *well-formed formulas* (wffs), which are finite sequences of symbols that are constructed from S using well-defined rules called *formation rules*. The collection of all wffs that can be constructed from S using the formation rules is called a *language on* S, and is denoted by L_S. The proof theory of a logic defines the expressions that are *derivable* (or *provable*) in the logic. It consists of: a subset of wffs called *axioms*, and a finite set of relations $\mathcal{R}_1, \mathcal{R}_2 \ldots, \mathcal{R}_n$ on wffs, called *inference rules*. An inference rule R_i maps a non-empty set of wffs, called the *premises* of R_i, to a single wff, called the *conclusion* of R_i. A *proof* is a sequence of wffs $\mathcal{A}_1, \mathcal{A}_2 \ldots, \mathcal{A}_m$ such that each $\mathcal{A}_i$ is either an axiom or a consequence of some of the preceding wffs by virtue of one of the inference rules. A *theorem* is a wff $\mathcal{A}$ such that there is a proof, the last wff of which is $\mathcal{A}$. Such a proof is called a proof of $\mathcal{A}$. The meta symbol $\vdash$ is used to represent provability, *i.e.* $\vdash \mathcal{A}$ states that $\mathcal{A}$ is a theorem. $\Gamma \vdash \mathcal{A}$ represents the derivation (or deduction) of the wff $\mathcal{A}$ from the set of wffs Γ, which is called the *assumptions* of $\mathcal{A}$. In the model theory of a logic, meanings are assigned to the expressions of the logic and then the expressions are checked to see if they are true or false. *Interpretations* give meaning to all symbols and, subsequently, to all wffs in the language of the logic. Based on the values assigned by an interpretation to the variables of a wff, the wff is said to be true or false. A wff that is true in all interpretations is called *logically valid* while a wff that is false in all interpretations is called *unsatisfiable*. The meta symbol $\models$ is used to represent validity, *i.e.* $\models \mathcal{A}$ states that $\mathcal{A}$ is logically valid. The logic is said to be *sound* if every theorem in it can be shown to be logically valid, while it is said to be *complete* if every logically valid wff in it can be derived as a theorem. The logic is said to be *consistent* if either $\vdash \mathcal{A}$ or $\vdash \neg\mathcal{A}$ (that is, not $\mathcal{A}$) but not both, *i.e.* you cannot prove both a wff and its negation. More details on deductive systems can be found in [1].

2.2. First Order Logic (FOL)

First-Order Logic is the most familiar logic to mathematicians. By embedding *proper* axioms, which are axioms that are defined on a specific domain, in FOL, we obtain other first-order theories such as *number theory* and *group theory*. As stated in the previous section, FOL has both proof and model theories. However, since ATP is concerned only with proofs, and hence the proof theory, we will only describe the proof theory of FOL.

The Alphabet of FOL

1. A set of constant symbols such as a, b, $c, \dots$ (may be followed by subscripts). Constant symbols can also be from some domains such as integers or strings, *e.g.* 0, 1, 'smith', 'dog', *etc.*
2. A set of variable symbols X, Y, $Z, \dots$ (may be followed by subscripts). Variable symbols can also be from other domains, *e.g.*, 'book', *etc.*
3. A set of function symbols $f(x)$, $g(y,z), \dots$ (may be followed by subscripts). Function symbols can also be from other domains, *e.g.*, successor(5), father(jim).
4. A set of predicate symbols P, Q, $R, \dots$ (may be followed by subscripts). Predicate symbols can also be from other domains, *e.g.*, equal(3,1+2), office (jim,243).
5. A set of logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not), and $\rightarrow$ (imply).
6. A set of punctuation symbols: '(', ')', and ','.
7. The quantifier symbols $\forall$ (for all) and $\exists$ (there exists).

The number of arguments of a function or a predicate symbol is called its *arity*. *i.e.*, $f(X,Y)$ has an arity 2 while $P(X,Y,a)$ has an arity 3.

The Formation Rules of FOL

The smallest expression in FOL is the *term*. A term is a constant symbol, a variable symbol, or a function symbol of arity n applied to n terms. An *atomic formula* is a term or a predicate symbol of arity n applied to n terms. The set of *well-formed formulas* (wffs) of FOL is inductively defined as follows:

1. Every atomic formula is a wff.
2. If $\mathcal{A}$ and $\mathcal{B}$ are wffs then so are:

   $(\neg\mathcal{A})$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, $(\mathcal{A} \rightarrow \mathcal{B})$, $(\forall x \mathcal{A})$, and $(\exists x \mathcal{A})$.

Given a wff $\forall x \mathcal{A}$ (or $\exists x \mathcal{A}$), $\mathcal{A}$ is called the *scope* of the quantifier $\forall x$ (or $\exists x$). An occurrence of a variable x is *bound* in a wff $\mathcal{A}$ if it is either the occurrence of x in a quantifier $\forall x$ (or $\exists x$) in $\mathcal{A}$, or it lies within the scope of a quantifier $\forall x$ (or $\exists x$). A variable is said to be *free* if it is not bound. If $\mathcal{A}$ is a wff and t is a term of FOL then t is *free for x_i in $\mathcal{A}$* if x_i does not occur free in $\mathcal{A}$ within the scope of a quantification $\forall x_j \mathcal{B}$, (or $\exists x_j \mathcal{B}$), where x_j is any variable occurring in t.

The Axioms of FOL

For any wffs $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ in FOL, the following are the (*logical*) axioms of FOL [1]:

1. $\mathcal{A} \rightarrow (\mathcal{B} \rightarrow \mathcal{A})$.
2. $(\mathcal{A} \rightarrow (\mathcal{B} \rightarrow \mathcal{C})) \rightarrow ((\mathcal{A} \rightarrow \mathcal{B}) \rightarrow (\mathcal{A} \rightarrow \mathcal{C}))$.
3. $(\neg\mathcal{B} \rightarrow \neg\mathcal{A}) \rightarrow ((\neg\mathcal{B} \rightarrow \mathcal{A}) \rightarrow \mathcal{B})$.

4. $\forall x_i \mathcal{A}(x_i) \to \mathcal{A}(t)$ if $\mathcal{A}(x_i)$ is a wff with free variable x_i, and t is a term which is free for x_i in $A(x_i)$, t can be identical to x_i.
5. $\forall x_i(\mathcal{A} \to \mathcal{B}) \to (\mathcal{A} \to \forall x_i \mathcal{B})$ if $\mathcal{A}$ does not contain any free occurrence of x_i.

The Inference Rules for FOL

1. From $\mathcal{A} \to \mathcal{B}$ and $\mathcal{A}$ conclude $\mathcal{B}$. "*Modus Ponens*, or *MP*"
2. From $\mathcal{A}$ conclude $\forall x_i \mathcal{A}$. "*Generalization*, or *GEN*".

An Example of A First Order Theory

Group theory is an example of a first-order theory. This theory has one constant symbol **1**, one function symbol **product(t, s)** (this function notation will be written as t^*s) where t and s are terms, and one predicate symbol **equal(t, s)** (this predicate notation will be written as $t = s$). In addition to the FOL logical axioms this theory has the following proper axioms:

Ax.1) $\forall x_1 \forall x_2 \forall x_3 (x_1 * (x_2 * x_3) = (x_1 * x_2) * x_3)$	(Associativity)
Ax.2) $\forall x_1 (1 * x_1 = x_1)$	(Identity)
Ax.3) $\forall x_1 \exists x_2 (x_2 * x_1 = 1)$	(Inverse)
Ax.4) $\forall x_1 (x_1 = x_1)$	(Reflexivity of =)
Ax.5) $\forall x_1 \forall x_2 (x_1 = x_2) \to (x_2 = x_1)$	(Symmetry of =)
Ax.6) $\forall x_1 \forall x_2 \forall x_3 ((x_1 = x_2 \land x_2 = x_3) \to x_1 = x_3)$	(Transitivity of =)
Ax.7) $\forall x_1 \forall x_2 \forall x_3 ((x_2 = x_3) \to x_1 * x_2 = x_1 * x_3 \land x_2 * x_1 = x_3 * x_1)$	(Substitutivity of =)

From the above axioms we can derive the following proof of the theorem:

$$(x_1 * x_1 = 1) \to (x_2 * x_1 = x_1 * x_2) \text{ for any } x_1 \text{ and } x_2$$

(1)	$x_1 * x_1 = 1$	(Assumption)
(2)	$x_2 * x_2 = 1$	(Assumption)
(3)	$(x_1 * x_2) * (x_1 * x_2) = 1$	(Assumption)
(4)	$x_1 * ((x_1 * x_2) * (x_1 * x_2)) = x_1 * 1$	(From Ax.7 and 3)
(5)	$x_1 * ((x_1 * x_2) * (x_1 * x_2)) = x_1$	(From Ax.2 and 4)
(6)	$(x_1 * (x_1 * x_2)) * (x_1 * x_2) = x_1$	(From Ax.1 and 5)
(7)	$((x_1 * x_1) * x_2) * (x_1 * x_2) = x_1$	(From Ax.1 and 6)
(8)	$(1 * x_2) * (x_1 * x_2) = x_1$	(From 1 and 7)
(9)	$x_2 * (x_1 * x_2) = x_1$	(From Ax.2 and 8)
(10)	$(x_2 * (x_1 * x_2)) * x_2 = x_1 * x_2$	(From Ax.7 and 9)
(11)	$x_2 * ((x_1 * x_2) * x_2) = x_1 * x_2$	(From Ax.1 and 10)
(12)	$x_2 * (x_1 * (x_2 * x_2)) = x_1 * x_2$	(From Ax.1 and 11)
(13)	$x_2 * (x_1 * 1) = x_1 * x_2$	(From 2 and 12)
(14)	$x_2 * x_1 = x_1 * x_2$	(From Ax.2 and 13)

3. INTRODUCTION TO AUTOMATED THEOREM PROVING

As shown in the previous section a wff $\mathcal{A}$ can be proved to be a theorem in some logic $\mathbf{L}$ if it can be derived from the axioms (and theorems) of $\mathbf{L}$ using its inference rules. If the proving process can be done mechanically using a computer program then that program is called an *automated theorem prover.* A more suitable definition of ATP is given in [2]:

"Automated Theorem Proving (ATP) is concerned with the task of mechanizing mathematical (or logical) reasoning. Proofs of mathematical theorems are performed by a computer, analogously to the way arithmetical problems are solved by a calculator".

In this section we present an overview of ATP. We begin with a historical background of the evolution of the field. We then include the basic terminology and notation used in ATP, followed by a description of the major elements of a typical theorem prover.

3.1. Historical Background

Mechanical theorem proving developed as a spin-off from the study of formal languages, such as First Order Logic (FOL). These languages were used by researchers to state rigorously a wide range of problems in an unambiguous way [3, 4]. Later, logicians formulated inference rules that enabled them to draw sound and correct conclusions from given statements. These new conclusions were called *theorems*, and the sequence of steps followed to draw a theorem was called a *proof.* When digital computers were invented, logicians became interested in using them to prove theorems automatically; *i.e.*, by giving the theorem to the computer and waiting until the computer proved the theorem without intervention from the user. These were called *stand-alone provers.* Some of the provers developed at that time were the Logic Machine by Newell, Simon, and Shaw [5], and the Geometry Machine by Hao Wang and Gelerntner [6]. The way a theorem was proved was actually by applying inference rules to axioms to generate theorems, which were also used in generating new theorems and so on until, eventually, the desired theorem was derived. The collection of all generated theorems was called the *search-space.* Since a search-space was always large it was required to develop some mechanisms to control it. These were known as *control strategies.* In the late 1950s and early 1960s, the concentration was based on using FOL as the main methodology of research. The development of *resolution* (an inference rule) in 1964 [7] was a major change in the field. In the late 1960s and early 1970s, researchers advanced the field by defining more inference rules, developing some search-space control strategies, and extending the applications to fields other than Mathematics [4, 8, 9]. *Interactive provers* followed, which initially produced part of the required proof, then returned the partial information to the user who then could direct/redirect the proof [10–12]. A great deal of success has been achieved recently. Several theorem provers are available today. Some of these provers are general, others are designed for specific applications, such as electronic circuit design, expert systems, program verification, and formal logic [13, 14]. There is, however, much more research to be done. Controlling the search-space is still a major problem. The development of theorem provers that are general and effective has not yet been accomplished. Effective provers cannot prove all theorems, and general provers are inefficient, consuming vast amounts of space and time [8, 15].

3.2. Terminology and Notations

Now we will define some ATP notations and terminology, some of which are derived from FOL while the others are specific to ATP. As previously defined in Section 2.2, a *term* is a constant symbol, a variable symbol, or a function symbol of arity n applied to n terms. An *atomic formula* is a term or a predicate symbol of arity n applied to n terms. A *literal* is an atomic formula (or the negation of an atomic formula), and it is called a *positive* (or a *negative*) literal. A *clause* is the disjunction ($\vee$) of a (finite) set of literals with no literal appearing twice. A clause with no literals is called the *empty* clause and is denoted by $\square$. A *Horn* clause is a clause that has at most one positive literal. If all the literals in a clause are positive (negative) then the clause is called a *positive* (*negative*) clause. A clause with both positive and negative literals is called a *mixed* clause. A clause that has a single literal is called a *unit clause*. A clause with more than one literal is called a *non-unit* clause. A wff in a clausal form has the form:

$$\forall x_1 x_2 \ldots x_n (C_1 \wedge C_2 \wedge \ldots \wedge C_m)$$

where each C_i is a clause, and $x_1, x_2, \ldots x_n$ are all the distinct variables occurring in C_i's. A substitution θ (of terms for variables) is a set $\{t_1 \rightarrow x_1, t_2, \rightarrow x_2 \ldots, t_n \rightarrow x_n\}$ where each x_i is a distinct variable and each t_i is a term which is not identical to the corresponding x_i. If l is a literal then $l\theta$ is the result of applying the substitution θ on the variables of l. A substitution q is a *unifier* for the literals $l_1, l_2 \ldots l_n$ if $l_1\theta = l_2\theta = \ldots l_n\theta$, and the literals $l_1, l_2, \ldots l_n$ are called *unifiable*. *Resolution*, sometimes called *binary resolution*, is an inference rule applied to two clauses. Resolution selects unifiable literals in the two clauses of the same predicate but of an opposite sign and yields a clause, called the *resolvent*. For example, if we have the following clauses:

C1: $P(x) \vee Q(x,a) \vee R(x,y)$ and

C2: $\neg P(a) \vee R(y,z)$

then applying resolution on them yields the following resolvent clause after eliminating $P(x)$ and $\neg P(a)$ and substituting a for x:

C3: $Q(a,a) \vee R(a,y) \vee R(y,z)$.

More details can be found in [1, 2].

3.3. Major Elements of a Theorem Prover

Although theorem provers may vary in their components, most of them share the following major elements:

1. *Information Representation Language*:

   The automated theorem prover must have a language that is capable of representing the information relating to the problem at hand. This language should be able to represent the required facts, relationships and concepts in a clear and unambiguous manner. Some of the existing languages are: the *Clause language* in which FOL wffs are converted to clauses [9], *Matrix representation* in which wffs are represented in a special tabular form [16, 17], *Non-Clausal representation* [18], and *Natural Deduction representation* [10, 12] in which wffs are treated in their original form.

2. *Inference Rules*:

   The prover also needs some inference rules which enable it to draw conclusions. A single inference rule, at least in practice, will not suit all kinds of applications. An example is resolution which can be applied to any problem if that problem can be represented as a set of clauses. However, it has the disadvantage that its deduction steps are too small, *i.e.*, it can process only two clauses at a time. *Hyperresolution* and *UR-resolution* [15] rules are similar to resolution except that they may work with two or more clauses at the same time. There are also some modified versions of resolution which are applied to statements not in a clause form, such as the *Non-Clause resolution* [18]. Natural deduction theorem provers use inference rules that manipulate the wffs by introducing and/or eliminating their connectives and/or quantifiers [10, 12].

3. *Control Strategies*:

   For a theorem prover to be effective, it should have some strategies to direct the application of inference rules towards an appropriate direction, and some strategies which restrict their applications to the problem domain. This is necessary since applying the inference rules without control may generate a lot of irrelevant information which may cause problems in terms of time or space limitation [9]. Some of these strategies are:

   (*a*) The weighting strategy. In this strategy, the user provides the program with some values or *weights* for the concepts or symbols of the problem at hand. The program uses these weights to determine which part of the problem to consider next. This makes the program reach the solution of the problem in a more efficient way, provided that the given weights are correct.

   (*b*) The set of support strategy. In this strategy, the user provides the program with a particular set of the input parameters, called the *set of support* for the problem at hand. When the program starts its deduction process it considers these parameters as the base of the search for the needed answer. The remaining input parameters are used only if they are needed to complete a deduction step.

In addition to the above elements, some of the existing theorem provers may have the following elements [9, 14]:

1. *Demodulation Procedures*:

   These procedures rewrite some of the information into a canonical form before using them in the reasoning process. For example, if the system has the modulator *equal(prod(1,X),X)*, and the program generates the clause *gt(prod(1,Y),Z)*, then the program automatically rewrites it as *gt(Y,Z)* before considering it in any further reasoning

2. *Subsumption Procedures*:

   These procedures eliminate any information that can be captured by other existing information. For example, if the program has the clause *equal(prod(X,1),X)* and if the clause *equal(prod(5,1), 5)* is generated later during the reasoning process, then the system will discard the new clause, since it is already subsumed by the first clause.

4. THE CURRENT STATUS OF ATP

4.1. Types of Proving Methodologies

Automated theorem provers can determine if a wff $\mathcal{A}$ is a theorem, in some logic, using one of the following approaches:

1. *The Refutation Approach*:

   This approach is based on the following concept: A wff $\mathcal{A}$ is a theorem if and only if it is logically valid; $\mathcal{A}$ is logically valid if and only if $\neg\mathcal{A}$ is unsatisfiable. Instead of showing that $\mathcal{A}$ is a theorem we may show that $\neg\mathcal{A}$ is unsatisfiable. This can be done by *refutation*. Refutation is a process in which the wff $\neg\mathcal{A}$ is added to the axioms of the theory, and then inference rules are applied to derive some contradiction. This contradiction indicates that $\neg\mathcal{A}$ is unsatisfiable, which proves that $\mathcal{A}$ is a theorem. Many theorem provers use refutation as their proving approach. They use various versions of resolution as their inference rules. An example of these provers is OTTER [9]. Prolog is also considered as a refutation theorem prover [19]. OTTER will be described in more detail in Section 4.4.

2. *The Direct Proving Approach*:

   In this approach $\mathcal{A}$ is proven as a theorem by deriving it from the axioms with the application of inference rules. These proofs can be done by *forward* or *backward* chaining. In forward chaining, inference rules are applied to the axioms to derive new theorems. These new theorems are used to derive additional theorems. This process continues until either $\mathcal{A}$ is derived (in which case it is a theorem), or until the time (or space) limit is exceeded. In backward chaining, the wff $\mathcal{A}$ is reduced to simpler wffs. These wffs are then reduced further and further. This process continues until all the new wffs are reduced to axioms (in which case $\mathcal{A}$ is a theorem), or until the time (or space) limit is exceeded. Some of the provers that use this approach are: the Boyer-Moore prover [13], IMPLY [10], and LCF [11, 12]. These provers will be described in detail in Section 4.4.

4.2. Application Fields of ATP

ATP is a rapidly growing field in computer science. There are many research areas that are currently under investigation [15]. These areas vary between improving the current status of the field, *e.g.* enhancing some of the existing inference rules, and/or developing new techniques that will produce better results. Another aspect of current research is *completeness* versus *effectiveness*. Completeness refers to the process of developing theorem provers which can solve problems from different areas, *i.e.*, general, although inefficiently, while effectiveness refers to the process of developing theorem provers which can solve specific problems, *i.e.*, special-purpose, but efficiently. There are several areas for which theorem provers have proved to be good and productive tools, but due to space limitation we will only outline some of the major application areas and the reader can refer to [2, 8, 9] for more details.

1. *Program Synthesis*:

   Program synthesis is the process of constructing programs directly from specification without using the classical approach of design, implementation and testing. One of the major approaches of program synthesis is based on theorem proving. In this approach the specification of the desired program is given as a mathematical formula. A theorem prover is used to prove that there is a program that satisfies that specification. The program itself is constructed as a side-effect of the proof.

   Manna and Waldinger have been working in this area since the early 70's [20]. Most of their work is based on resolution based theorem proving. A natural deduction-based approach is developed in [21]. Interested readers can find more details in [8, 22–24].

2. *Hardware Verification*:

   Hardware verification, in its simplest meaning, is the process of checking and verifying that a given hardware circuit is actually performing the functions for which it was implemented. This verification varies from a simple circuit to a more complicated one.

   This research area has gained more attention recently due to the fact that verifying complex hardware circuits by traditional methods is becoming more difficult and more expensive. Some of the existing systems that are mainly used in hardware verification are *LCF-LSM* [25] and *HOL* [26]. Some of the recent application of theorem provers in hardware verification is discussed in [27]. Wos *et al.* in [9] show many examples of how the OTTER system (described in Section 4.4) is used in logic circuit design and verification.

3. *Research in Mathematics*:

   Mathematics is the oldest research area in which theorem provers have been used. Since their early development ATP systems have been heavily used in proving mathematical theorems. These proofs are either new proofs of theorems not proven manually before, or proofs of old theorems, which confirm their old manual proofs [28]. IMPLY (described in Section 4.4) has been heavily used in proving theorems in the *set theory*, and *limit theory*. OTTER and its predecessors AURA [29] and ITP [14] have been heavily used in proving theorems from *Group theory*, *Ring theory*, and *Ternary Boolean algebra* [30].

4.3. ATP Current Research

In the previous section we discussed some of the application areas of ATP. In this section we briefly outline the general directions of research in this field. In the early development of ATP the main direction of research was to discover and find new techniques, inference rules, strategies, *etc.* that would enable theorem provers to work better. Although this is still an active area, see [31] for example, most of the current research is concentrated on the use of existing theorem provers in almost all different fields. For example, reference [32] describes how an automated theorem prover is used in prediction the behavior of mechanical devices. Reference [33] describes how to use theorem provers to search for a model of a solution for a problem. Most of the current research is reported in specialized journals such as the *Journal of Automated Reasoning*, see for example [34].

4.4. Examples of Automated Theorem Provers

1. *Boyer and Moore*

   This prover was developed by R. Boyer and J. Moore in the late 1970s to prove inductive theorems in the standard mathematical style [2, 13]. The main motivation behind the development of the system was to automate the proof process as far as possible. Boyer and Moore in [13] say that they were interested in how people go about proving theorems by induction. That is why their system has many heuristics for inventing induction, for removing undesirable elements from conjectures and for generalizing formulas.

   The language of the theorem prover is a version of pure Lisp, since initially the system was built with the intention of proving theorems about recursive functions. Lisp, stands for List Processing, is a computer language that is commonly used in Artificial Intelligence (AI) research. The system strategy is to simplify the given conjecture to prove it. If this simplification fails then the prover will invent an induction argument based on the axioms defining the types of objects possible (numbers, list, *etc.*,) and on the recursively defined functions appearing in the conjecture.

   The system and its underlying theory has been used in some applications such as program verification and logic circuit design. More details can be found in [2]. Recently some interactive enhancement has been added to increase the power of the system [35].

2. *IMPLY*

   Bledsoe and his working group at the University of Texas at Austin have developed an interactive theorem prover called UT. Although *IMPLY* is the central routine of this theorem prover, UT is known to many researchers as IMPLY [10].

   The prover is a natural deduction-based system that was used to prove theorems in first order logic, and some extensions of it. *Natural deduction* is a proving mechanism (or system) in which wffs are proven in their original form without being transformed to clause. It is called natural because it resembles the natural way of theorem proving. For more details on natural deduction see [2, 12, 36].

   IMPLY uses the concepts of sub-goaling, reduction (rewrite rules) procedures, controlled definition instantiation, controlled forward chaining, conditional rewriting and conditional procedures, algebraic specification, and induction.

The prover has an interactive user interface. The user can wait for the prover to prove the given theorem within a redefined time interval. If the prover fails then the user can interact and give it other directions. Although IMPLY is a general theorem prover, it has been used to prove theorems in the following areas:

- set theory,
- limit theorems,
- topology, and
- program verification.

3. *LCF*

LCF Stands for Logic for Computable Functions. LCF is a logic developed by Dana Scott in 1969 in which facts about recursively defined functions can be formulated and proved. The Stanford LCF theorem prover was developed by Milner to perform proofs in this logic. The proof was done step by step, *i.e.*, it is a proof checker. Milner developed a meta language, ML, in which the prover can be programmed, the result was Edinburgh LCF [11]. Cambridge LCF extended the logic of Edinburgh LCF with $\vee$, $\exists$, $\longleftrightarrow$(stands for *if and only if* or $\rightarrow \;\wedge\; \leftarrow$) and predicates, improved the efficiency and added several inference mechanisms [12].

LCF uses ML as its representation language. It uses the natural deduction connectives/quantifiers introduction and elimination inference rules. It also uses both forward and backward approaches for proving theorems. This theorem prover can be used in the following applications:

- Experimenting with first order proofs.
- Studying abstract constructions in domain theory.
- Comparing the denotational semantics of programming languages.
- Verifying functional programs.

LCF is currently used in many proving systems such as:

(*a*) LCF-LSM and HOL: which were developed for verifying hardware [25, 26].

(*b*) Nuprl: which supports constructive type theory and is used heavily in mathematics [37].

4. *OTTER*

The *OTTER* (Organized Techniques for Theorem-proving and Effective Research) system was developed by W. McCune of the Mathematics and Computer Science Division at Argonne National Laboratory [9, 38]. It was developed as a result of the continuous success of the former automated theorem provers AURA and ITP.

OTTER uses the Clause language as its representation language, in addition to FOL wffs. Its inference rules include various versions of resolution, *e.g.* hyperresolution. It also uses various control strategies such as the set-of-support, and demodulation and subsumption procedures. It has been successfully used in the following areas:

- Logic circuit design and validation.
- Program verification.
- Formal logic
- Mathematics.

OTTER is written in C and therefore it is very portable. It is available on PC-DOS, Macintosh, and UNIX platforms.

5. A THEOREM PROVER IN ACTION

So far we have described some theorem provers and their application areas. The reader can find in the corresponding references how these provers are actually used in solving problems. However, for the purpose of completeness it is important to show an example of a problem that is solved by a theorem prover. In this section we will outline a proof of a validation of a digital circuit that was obtained by the help of OTTER. The details of this proof can be found in [9].

5.1. The Problem

Prove that the output of the circuit shown in Figure 1, which consists of NAND gates, is equivalent to OR($i1$,$i2$), *i.e.* an OR gate.

5.2. The Solution

We will use the clause language of OTTER to represent the problem and its solution. All clauses will be numbered for ease of referencing. First of all we need to identify the symbols, axioms, and inference rules and demodulators (see Section 3.3) that will be used to prove the above claim. We will use the following symbols:

- the constant symbols $i1, i2$ to represent the input to the circuit, $o1$ to represent the output of the circuit, and $a1, a1$ to represent the intermediate output as shown in the figure.
- the variable symbols x, y to represent general variables.
- the predicate P to represent the output of the circuit.

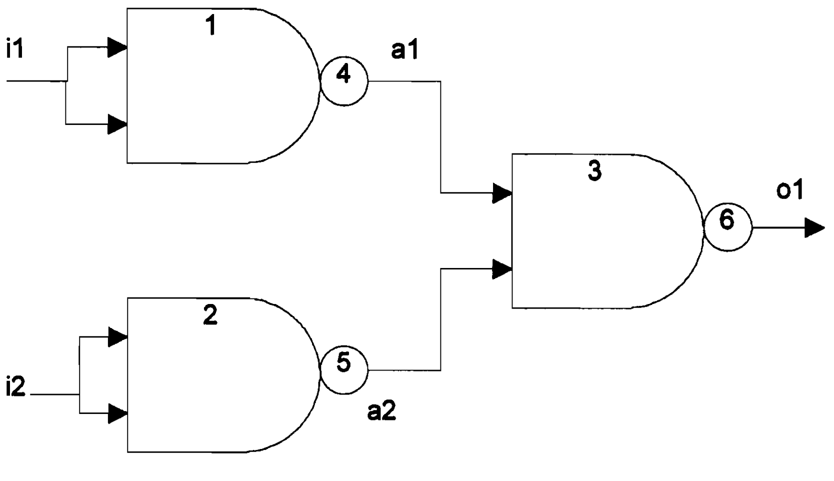

Figure 1. Nand-Or Equivalent Diagram.

From Figure 1 we can write the following demodulators:

(1) EQUAL($o1$,nand($a1, a2$))

(2) EQUAL($a1$,nand($i1, i1$))

(3) EQUAL($a2$,nand($i2, i2$)).

From the definition of logical circuits we can write the following demodulators:

(4)	EQUAL(nand(x, y),not(and(x, y)))	Relation between AND and NAND
(5)	EQUAL(and(x, x),x)	
(6)	EQUAL(and(not(x),not(y)),not(or(x, y)))	Relation between AND and OR
(7)	EQUAL(not(not(x)),x).	

The proof using the above demodulator is shown below:

(8)	P($o1$)	(The starting clause)
(9)	P(nand($a1, a2$))	(Applying 1 on 8)
(10)	P(nand($a1$,nand($i2, i2$)))	(Applying 3 on 9)
(11)	P(nand($a1$,not(and($i2, i2$))))	(Applying 4 on 10)
(12)	P(nand($a1$,not($i2$)))	(Applying 5 on 11)
(13)	P(nand(nand($i1, i1$),not($i2$)))	(Applying 2 on 12)
(14)	P(nand(not(and($i1, i1$)),not($i2$)))	(Applying 4 on 13)
(15)	P(nand(not($i1$),not($i2$)))	(Applying 5 on 14)
(16)	P(not(and(not($i1$),not($i2$))))	(Applying 4 on 15)
(17)	P(not(not(or($i1, i2$))))	(Applying 6 on 16)
(18)	P(or($i1, i2$))	(Applying 7 on 17)

6. CONCLUSION

People have been, will always be, anxious to use computers in every possible way that may ease their jobs, and mathematicians are no different. In this paper we have presented an overview of the field of automated theorem proving and showed the current status of this important area. Researchers are becoming extremely interested in this field. Some of them are interested in improving the tools that are used by theorem provers, such as improving search-space controlling strategies or developing more useful inference rules. Other researchers are interested in using these provers in different application areas such as program synthesis, verification, and debugging. With the advances of computer technology theorem provers have become available to all different kind of computer users. The availability of these provers on different platforms such as Personal Computers (PCs), Macintoshes, Workstations and bigger systems encourages more and more people to use them. We expect that all these factors will improve and advance this field rapidly and will produce different systems that will be used in a variety of applications

ACKNOWLEDGMENT

The author would like to thank the following KFUPM faculty members for their revision and suggestions on this manuscript: Dr. M. Najjar (Information and Computer Science), Dr. S. Sait (Computer Engineering), and Drs. J. Ahsan and A. Chaudhry (Mathematical Sciences). Dr. M. Al-Bar (Mathematical Sciences) has contributed with some mathematical proofs. A special thanks goes to Dr. L. Wos of Argonne National Laboratory, Illinois, USA for providing some of the recent references.

REFERENCES

[1] E. Mendelson, *Introduction to Mathematical Logic.* Belmont, CA: Wadsworth & Brooks/Cob. Advanced Book & Software, 1987.

[2] D. Duffy, *Principles of Automated Theorem Proving.* New York: John Wiley & Sons, 1991.

[3] M. Davis, "The Pre-History and Early History of Automated Deduction", in *Automation of Reasoning, Volume 1.* ed. J. Siekman and G. Wrightson. Springer-Verlag, 1983, pp. 1–27.

[4] L. Wos and L. Henschen, "Automated Theorem Proving 1965–1970", in *Automation of Reasoning, Volume 2.* ed. J. Siekman and G. Wrightson. Springer-Verlag, 1983, pp. 1–24.

[5] A. Newell, J. C. Shaw, and H. A. Simon, "Emperical Explorations of the Logic Theory Machine: a Case Study in Heuristics", *Western Joint Computer Conference,* 1956, pp. 218–239.

[6] H. Gelerntner, "Realization of Geometry Theorem-Proving Machine", *International Conference on Information Processing,* 1959, pp. 273–282.

[7] J. Robinson, "A Machine Oriented Logic Based on the Resolution Principle", *Journal of the ACM,* **12(1)** (1965), pp. 23–41.

[8] R. Boyer, J. Moore, W. Bledsoe, L. Henschen, B. Buchanan, G. Wrighstone, L. Wos, F. Pereira, and C. Green, "An Overview of Automated Reasoning", *Journal of Automated Reasoning,* **1(1)** (1985), pp. 5–48.

[9] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1992.

[10] W. Bledsoe, "The UT Prover", *Math Department Memo ATP-17B, University of Texas at Austin,* 1983.

[11] A. Milner, M.J.C. Gordon, and C. Wadsworth, "Edinburgh LCF", *Springer-Verlag Lecture Notes in Computer Science,* vol. 78, 1979.

[12] L. Paulson, *Logic and Computation: Interactive Proofs with Cambridge LCF.* Cambridge, U.K.: Cambridge University Press, 1987.

[13] R. Boyer and J. Moore, *A Computational Logic Handbook.* Boston: Academic Press, 1988.

[14] E. Lusk and R. Overbeek, "The Automated Reasoning System ITP", *ANL-84-27.* Argonne, IL: Argonne National Laboratory, 1984.

[15] L. Wos, *Automated Reasoning: 33 Basic Research Problems.* Englewood Cliffs, N.J.: Prentice-Hall, 1988.

[16] P. Andrews, "Theorem Proving *via* General Matings", *Journal of the ACM,* **28(2)** (1981), pp. 193–214.

[17] W. Bibel, "On Matrix with Connections", *Journal of the ACM,* **28(4)** (1981), pp. 633–645.

[18] N. Murray, "Completely Non-Clausal Theorem Proving", *Artificial Intelligence,* **18(1)** (1982), pp. 67–85.

[19] M. Clocksin and M. Melish, *Programming in Prolog.* New York: Springer-Verlag, 1987.

[20] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis", *ACM Transactions on Programming Languages and Systems,* **2** (1980), pp. 90–121.

[21] T. Maghrabi, "The Synthesis Tableau: A Deductive Framework for Program Synthesis Based on Sequent Calculus", *PhD Thesis, Arizona State University, Tempe, Arizona, U.S.A.,* August 1992.

[22] N. Dershowitze, *The Evolution of Programs.* Cambridge: Birkhauser Boston, Inc., 1983.

[23] G. Dromey, *Program Derivation: The Development of Programs from Specifications.* Sydney, Australia: Addison-Wesley, 1989.

[24] R. Olsson, "Inductive Functional Programming Using Incremental Program Transformation", *Artificial Intelligence*, **74** (1995), pp. 55–81.

[25] M.J.C. Gordon, "Lcf-lsm: A System for Specifying and Verifying Hardware", *Report 41, Computer Laboratory, University of Cambridge*, 1983.

[26] M.J.C. Gordon, "Hol: A Proof Generating System for Higher-Order Logic", *Report 103, Computer Laboratory, University of Cambridge*, 1987.

[27] D. Sarkar, I. Chakrabarti, and A. Majumdar, "Identification of Inductive Properties During Verification of Synchronous Sequential Circuits", *Journal of Automated Reasoning*, **14(3)** (1995), pp. 427–462.

[28] X. Gao, S. Chou, and J. Zhang, "Automated Generation of Readable Proofs with Geometric Invariants *i* and *ii*", *Journal of Automated Reasoning*, **17(3)** (1996), pp. 349–370.

[29] E. Lusk, L. Wos, R. Overbeek, and J. Boyle, *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.

[30] R. Padmanabhan and W. McCune, "Single Identities for Ternary Boolean Algebras", *Computers and Mathematics with Applications*, **29(2)** (1995), pp. 13–16.

[31] R. Veroff, "Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case studies", *Journal of Automated Reasoning*, **16(3)** (1996), pp. 223–239.

[32] A. Gelsey, "Automated Reasoning About Machines", *Artificial Intelligence*, **74** (1995), pp. 1–53.

[33] S. Lee and D. Plaisted, "Problem Solving by Search for Models with a Theorem Prover", *Artificial Intelligence*, **69** (1994), pp. 205–233.

[34] L. Wos, "The Power of Combining Resonance with Heat", *Journal of Automated Reasoning*, **17(1)** (1996), pp. 23–81.

[35] M. Kaufmann, R. Boyer, and J. Moore, "The Boyer–Moore Theorem Prover and its Interactive Enhancement", *Computers and Mathematics with Applications*, **29(2)** (1995), pp. 27–62.

[36] J. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row Publishers, Inc., 1986.

[37] H. Bromley, W. Cleavland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mandler, P. Panangaden, J. Sasaki, R. Constable, S. Allen, and S. Smith, *Implementing Mathematics with The Nuprl Proof Development System.* Prentice-Hall, 1986.

[38] L. Wos, *The Automation of Reasoning: An Experimenter's Notebook with OTTER Tutorial.* London: Academic Press, 1996.

Paper Received 1 October 1994; Revised 29 August 1995; Accepted 6 November 1995.