Automated reasoning

A valuable tool in validating our work

Automated reasoning is the attempt to prove statements with a computer in a law-like way. Applications include mathematical theorem proving; circuit design, validation, and diagnosis; program verification and validation; expert systems, and term rewriting systems.

Automated reasoning requires a very precise representation for the expressions being reasoned over. One such representation is the first-order predicate calculus. Expressions in this language are formed from predicates, arguments, quantifiers, and variables of quantification.

A theorem prover is an automated reasoning program which tries to determine if a sentence follows logically from a set of axioms. If it does, it is considered a theorem. Most theorem provers are based on unification and the resolution principle. Given a set of clauses, they try to determine whether a set of clauses, which include the axioms and negation of what we are trying to prove, is unsatisfiable, that is, not true under any interpretation. Such unsatisfiability is shown by deriving the empty clause via resolution. Proving theorems this way is called a resolution refutation.

In order for a computer to reason efficiently, the syntax must be simplified. That is, the predicate calculus expressions must be translated into a form that allows the use of a single inference rule known as the resolution principle.

The two most desirable features of an inference procedure are soundness and completeness. Soundness means that the inference procedure never draws false conclusions from true premises, and completeness means that the inference method is capable of drawing all possible conclusions which follow from the premises.

Daniel P. Murphy
Christopher J. Merz

A third feature which is extremely desirable, but can never be attained by any inference procedure for the predicate calculus, is decidability. An inference procedure is decidable if an effective procedure exists for determining if an arbitrary sentence follows from a given set of premises. Saying that no such decidable procedure exists for the predicate calculus is the same as saying there is no way of knowing if any proof procedure for the predicate calculus will ever terminate. Such proof procedures are called semi-decidable.

The basic resolution principle is sound, but not complete or decidable. It can be made complete by adding the feature of factoring. Losing completeness is a common result of trying to speed up resolution but is acceptable unlike the loss of soundness.

Diagnosis from first principles

One alternative to heuristic classification is diagnosis from first principles. Rather than relying on an expert's general rules (heuristics) about how a set of symptoms are usually associated with a certain fault(s), diagnosis from first principles relies on a device description to reason out how a device actually works. Such an approach eliminates the difficult task of eliciting knowledge from an expert. Also, it only requires a detailed description of how the device should behave.

One example of diagnosis from first principles is Reiter's theory where the diagnosis of a faulty device is made based on the *system description* (SD) of a device featuring a finite set of *system components*, and a set of *observations* (OBS) of the device. SD and OBS are finite sets of sentences in first-order predicate logic where SD describes the system components and OBS describes the symptoms of the device. A *diagnosis* for (SD, COMPONENTS, OBS) is a minimal set of faulty COMPONENTS. A COMPONENT is only a member of this diagnosis set if considering it non-faulty would cause a contradiction in the collection of logical statements describing the system (SD), the symptoms (OBS), and the other COMPONENTS which are considered non-faulty.

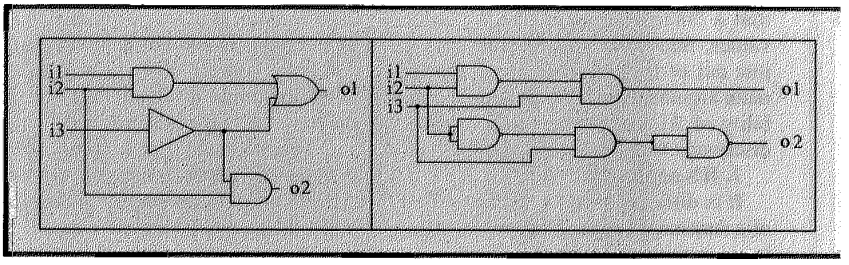

Fig. 1a *Fig. 1b*

Obviously, many different diagnoses of a faulty device could be constructed. One way of searching for diagnoses is to have an algorithm which generates possible diagnoses and tests them. A more efficient algorithm for finding diagnoses is made possible by considering the notion of a conflict set. A *conflict set* is a set of COMPONENTS {C1, C2, ..., Cn} which, if NOT considered faulty, causes a contradiction in the collection of logical statements describing the system (SD) and the symptoms (OBS). Reiter shows how to compute all diagnoses of (SD, COMPONENTS, OBS) given a theorem prover that will generate conflict

0278-6648/92/$3.00 © 1992 IEEE

sets for (SD, COMPONENTS, OBS). The key idea is to form *minimal hitting sets* which consist of a member(s) of each conflict set. The failure of the COMPONENTS in a hitting set would explain the faulty behavior of the device making the hitting set a possible diagnosis. Some diagnoses can be ruled out if they predict observations which have not actually been made.

The approaches to diagnosis from first principles developed so far are limited. They require a complete device description in order to be sound and complete. They assume connections between components of a device are working properly, and they rely on reasoning done over diagnosis theories formulated in full first-order logic which is only semi-decidable.

Logic circuit design and validation

We can also use automated reasoning to help us design and validate logic circuits. One problem circuit designers can face is to take a circuit specification in terms of ANDs, ORs, and NOTs and produce a circuit using only the more common, but less intuitive, gate NAND. For example, we might like to transform the circuit in Fig. 1a into the circuit in Fig. 1b which only uses NANDs.

We can do this by first defining our outputs in terms of our inputs: o1 = or(and(i1, i2), not (i3)) and o2 = and(not(i3),i2). Next, we define the relations between ANDs, ORs, and NOTs and NANDs with:

not(x) → nand(x,x)
or(x,y) → nand(not(x),not(y))
and(x,y) → not(nand(x,y))

We will also simplify our results with:

nand(nand(x,x),nand(x,x)) → x.

With these equalities and our original specification of our output, we use demodulation to produce the desired circuit containing only NANDs. Demodulation is the substituting or rewriting of one term by another equivalent one, such as not(x) by nand(x,x). Demodulators are applied when the first clause unifies with the term we are attempting to rewrite. The resulting term is the unifier applied to the second clause of the demodulator.

By repeatedly applying the demodulators to our given circuit we can produce circuits containing only NANDs such as in Figure 1b.

If, on the other hand, the specification is given as a table such as:

i2 \ i1	0	1
0	0	1
1	1	0

then we can represent it as table(i1, table(i2,0,1),table(i2,1,0)) where table(input,x,y) is interpreted as: if input is 1 return x else return y.

Using the demodulator table(input, x,y)→and(or(not(input),x)or(input,y)) and others listed in Figure 2 we can produce a circuit with ANDs, ORs and NOTs through demodulation. We can then also use the previous demodulators to put the circuit in terms of NANDs. Circuit validation is the process of assuring that a circuit performs as desired. For example, we might want to validate that the circuit nand(nand(x,y),nand(x,y)) is equivalent to and(x,y). Part of this is accomplished by using the previous equalities for NAND but rewriting in the other direction. Validation, however, is harder than circuit design where we can simply stop when all terms are NANDs. Terms in validation must have a canonical form to assure that all equivalencies are found. Subtle equalities such as or(x,or(x,y)) → or(x,y) must be added. In validation we must also assure that we never loop in trying to find our proof such as infinitely rewriting or(x,y) → or(y,x) → or(x,y) → ....

Program verification and validation

Computer software's growing complexity makes proving the correctness of the software extremely complicated. Traditionally, programs are proven "correct" by running the program on many sets of data and showing that the output is correct. Data is chosen to represent normal values and values at the extremes. The choosing of this data is not always straightforward; and, even when it is, this is not enough to show that the software is correct for all data. We can instead attempt to formally prove program correctness using automated reasoning techniques.

not(0) → 1	or(1,x) → 1
not(1) → 0	or(x,0) → x
and(x,1) → x	or(0,x) → x
and(1,x) → x	and(x,not(x)) → 0
and(x,0) → 0	and(not(x),x) → 0

Fig. 2

A procedure is proved correct if all inputs satisfying the input assumptions yield results satisfying the exiting requirements. Note that program correctness implies nothing more than this. Thus, the burden is placed on the programmer to give the complete specifications for a procedure. If an incomplete specification is given, then the proof that the procedure is "correct" means only that the procedure meets the specification given. It does not necessarily mean that the procedure produces the desired result.

For example, if we give the exit condition that an absolute value procedure returns a value greater than or equal to zero, we are not being complete. A function that always returns "1" meets that specification. We obviously need to add that the output is equal to the input or the negative of the input.

Hantler and King give a method for symbolic execution of a procedure. Symbolic execution does not use exact values, such as testing when x=3, but instead keeps track of variables and how they are manipulated. In their method, constraints are added to variables as warranted by execution of the procedure and all possible paths are considered. For example, if the statement

```
IF x<O THEN
y=3*z
ELSE
y=2*x
```

is executed, then two branches of execution must be considered with x<O as a constraint on one branch and x>=O on the other. Also y will be modified accordingly for each branch.

The symbolic execution of assignment statements is handled in the obvious way. Statements are executed similar to "while" statements except that the enclosed statements may be executed any number of times. To take

this into consideration, we add a "cut" statement to the beginning of the loop. This cut statement has a specification which acts as both input and output requirements. We test that the value coming in meets these specifications and continues to do so after the loop is performed. If this is shown, we know no matter how many times the loop is performed the requirements will be met.

Procedure calls could be handled by inserting the code where it is called from (and renaming variables) but this would mean repeatedly verifying the same code if it were called several times in the program. Instead we prove correctness once for the procedure separately and then use an abbreviated procedure at all other times. The procedure checks input assumptions, adds its exit conditions to the constraints, and gives new symbols to modified variables.

Automated reasoning is used in proving a program's correctness by having demodulation rules defining how each type of statement affects the program state (variable values, position in the program, and current constraints). We then type in our program with the input assumptions and our demodulators work on it from the initial state trying to prove that all exit conditions are met. If this happens, the program is proved correct.

Automated reasoning can be applied to many areas which most often have a common need: solving problems with unification and a tedious and repetitious proof procedure. The field is expanding and is being used either as integral or peripheral parts of different fields.

Read more about it

- Hantler, S. and King, J. (1976). "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, September 1976, pp331-351.
- Jackson, P. (1990). *Introduction to Expert Systems*, Addison Wesley.
- Reiter, R. (1987). "A Theory of Diagnosis from First Principles," Artificial Intelligence, Volume 32, pp. 57-95.
- Robinson, J. A., (1965). "A Machine-oriented Logic Based on the Resolution Principle," Journal of the Association for Computing Machinery, Volume 12, pp. 23-41.
- Wos, L., Overbeek, R., et. al., (1984). *Automated Reasoning: Introduction and Applications*, Prentice-Hall, Inc.
- Wos, L., (1988). *Automated Reasoning: 33 Basic Research Problems*, Prentice-Hall, Inc.

About the authors

Daniel P. Murphy recently received his Ph.D. in computer science at the University of Missouri-Rolla. His research interests include artificial intelligence, automated reasoning, and expert systems. Christopher J. Merz is a Ph.D. student in computer science at the University of California-Irvine. His research interests include neural networks and machine learning. ■