Model Checking

David L. Dill
Computer Systems Laboratory
Stanford University

Why isn't formal verification used routinely?

- Engineers don't know enough math/logic
- People aren't properly trained
- Bad notation and user interfaces
- Inertia – design methodologies are not easily changed
- Managers don't require it
- Blind stupidity

The main reason

Economics!

- In most cases, cost vs. benefit is unfavorable, or unproven.
- "Conventional" methods are extremely labor-intensive.
- Furthermore, the labor must be highly skilled.
- In many cases, use of formal methods would *delay* completion of the design.

Design time is often crucial for safety- and life-critical applications.

Most designs are "time-to-market" critical (e.g. cost of delay for an microprocessor release $10 million/week).

Model checking (finite-state methods)

Most NASA work focusses on very general theorem-proving methods.

Alternative: Use less general, but more automatic, methods.

In the finite-state domain, most verification problems become decidable.

Verification can be done automatically by implicit or explicit state enumeration.

Model-checking — applications

Finite-state methods have been applied successfully in several domains:

- Protocols (communication, network, cache coherence)
- Hardware state machine comparison
- Hardware controllers
- Asynchronous circuits

There are many success stories about using finite-state verification tools.

In some cases, they have been incorporated into commercial product design methodology.

Tradeoffs

Advantages of model-checking over theorem-proving:

- Highly automated
- Excellent at diagnosing design errors

Disadvantages of finite-state methods:

- Computational complexity
- Abrupt failure when problem grows
- Limited expressiveness

In reality, model-checking and theorem-proving are complementary techniques.

Explicit state enumeration

Basic idea: Generate reachable states "on-the-fly," searching for "bad" states.

"bad states" could be *deadlock*, or could violate a *per-state property* (e.g. mutual-exclusion).

Explicit state search

Behavioral description of the system must give *start states*, and a *next-state generator*, which maps a state to a set of next states.

Generators can be derived from almost any reasonable operational description.

Basic search procedure maintains

- a queue of states to be searched
- a table of states already searched

Search algorithm

While queue is not empty, remove a state s from it.
    Let Q be the set of next states of s.
        For each $s' \in Q$
            If s' is not in the state table
                If it is bad, report error and halt
                else enter s' in table and insert in the queue.

It is important to stop when error occurs (to avoid generating all states).

When an error occurs, a “counterexample” (path from start state to bad state) can be printed.

Search can be in any order, although breadth-first gives the shortest counterexamples.

Specifications

- Fixed properties (e.g. deadlock, “unspecified reception”)
- Comparison with another description (simulation, bisimulation, language inclusion)
- Temporal logic model checking (especially CTL)

State explosion problem

Of course, a small behavioral description may give rise a large state space.

This is known as the “state explosion problem.” It is the central problem in finite-state verification.

Most problems are at least PSPACE-complete, so a general worst-case solution is probably not available.

There are many methods that have been used to extend the practical boundaries of this method.

Simple methods

- Omit irrelevant implementation details
- Reduce size of system (“down scaling”)
- Focus on finding bugs, not proving correctness
- Use modular/compositional verification

Advanced methods

A number of sophisticated techniques for coping with the state explosion have emerged in the last few years.

Hash compaction — Verify probabilistically by storing a certificate for each state (down to 1 bit), to minimize table size.

Partial-order methods — Exploit independent events to reduce number of interleavings that must be modelled.

Symmetry reduction — Avoid redundancy from symmetry in the system.

Symbolic state exploration — Represent state space symbolically (using Boolean functions, linear inequalities).

BDD-based verification

BDD-based verification has been so successful that it deserves special attention.

A BDD is a data structure for representing Boolean functions.

Every state is represented as a bit-vector

State is in set iff Boolean function is true.

Breadth-first search can be done using BDD operations.

Sometimes, astronomically large state spaces can be checked with BDDs (Clarke: 10^{1300} states).

Sometimes, BDD-based verifiers are worse than explicit methods.

Verification Systems

Verification systems based on state enumeration have been around at least since 1981.

Some existing systems: SPIN (Holzmann et al. AT&T), COSPAN (Kurshan et al., AT&T), Murϕ (Dill, et al. Stanford), SMV (Clarke and McMillan, Carnegie-Mellon U.)

There are many success stories of using these systems to verify parts of protocols and hardware designs.

All of these systems are currently in use in industry.

Future directions

There are many exciting research topics in this area:

- New ideas for avoiding the state explosion
- Alternative symbolic representations
- Verification of real-time and hybrid systems.
- Methods supporting abstraction and refinement
- Extension to infinite-state systems.

Model checking $+$ theorem proving

One of the most exciting research areas is finding ways to combine finite-state and theorem-proving methods.

A simple idea is to use model-checking first to debug the problem and verify for small systems, then use theorem-proving for the general case.

Example: We described and verified a *distributed list protocol* (central part of a multiprocessor cache consistency algorithm) in Murϕ for 3 processes.

This helped us to debug the protocol *and verification conditions*.

Automatically translated Murϕ program to PVS, and verified for n process in PVS.

Model checking $+$ theorem proving

In some cases, tools can be complementary.

Example: We described the Sparc International's multiprocessor memory consistency models in Murϕ.

Murϕ model can be used to verify synchronization routines.

PVS used to verify consistency with a logical specification of the same memory model.

Integration

More ambitiously, finite-state methods can be incorporated into a theorem-proving framework.

Practical use of model-checkers already requires reasoning outside of the finite-state domain.

This reasoning can be a source of errors and omissions, and should be formalized.

Example: A μ-calculus model checker has been embedded in PVS.

When a user generates a lemma in μ-calculus, the model-checker can be called to check it automatically, much more efficiently than general decision procedures of PVS.

There are many opportunities for developing new reduction strategies.

Conclusions

- Economics are crucial.
- Use of verification for finding bugs may be more important than correctness proofs.
- Formal verification based on finite-state techniques is already successful, and is improving rapidly.
- Combining model-checking and theorem-proving may lead to necessary breakthroughs in verification productivity.