Model checking

1 November 2017

Verification purpose

show that program is *correct*
(if feasible)

finding *errors*
methods that only target error finding (testing)
or methods that try to prove correctness
and show error (counterexample) if they fail

Verification methods

static = without code execution

finding error patterns

dataflow analysis

formal verification

dynamic = by running code

instrumenting / running on virtual machine

symbolic execution (work with formulas, not values)

Trusting the verification outcome

A method is
sound ? = every answer is valid ?
complete ? = finds all the answers ?

Verification:
sound: a system reported as correct is correct
complete: can prove correctness of any system
  impossible for precise problems (e.g. halting)
  possible for more general ones (e.g. no type errors)

Error finding:
sound: every reported error is real
complete: finds all errors

Formal verification

Uses mathematical model of system
$\Rightarrow$ allows *guaranteed* (certified) results
within modeling assumptions (compiler, libraries, OS, hardware...)

Theorem proving
verification conditions (from Floyd/Hoare rules)
provers or satisfiability checkers (SAT-solvers)
may need human hints / annotations for complex cases
intense interaction with human expert

Model checking
system = finite-state automaton
algorithm = explore state space (graph traversal)
automated; gives counterexample in case of error
challenge: state space explosion

Model checking in brief

developed from 1981 (Clarke & Emerson; Sifakis – Turing award 2007)
initially applied to hardware and small concurrent programs

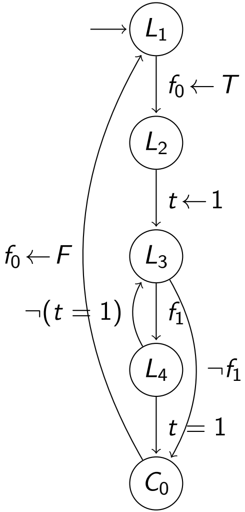
Example: Peterson's mutual exclusion algorithm

```
while (1) {
  L1: flag[0] = true; // try
  L2: turn = 1; // other's turn
  L3: while (flag[1] && turn==1)
    ; // wait
  C0: flag[0] = false;
}
```

```
while (1) {
  R1: flag[1] = true; //try
  R2: turn = 0; // other's turn
  R3: while (flag[0] && turn==0)
    ; // wait
  C1: flag[1] = false;
}
```

Can programs simultaneously reach critical section ?
labels C0 and C1, *before* setting to *false* (freeing resource)

Model checking: automaton representation



State space:

variables: 3 bits: f_0, f_1, t, initially $(?, ?, ?)$

program counters (2 threads)

$\Rightarrow$ cartesian product: pairs (pc_0, pc_1)

Explicit representation: $2^3 \cdot 5 \cdot 5$ states

Not all states are *reachable* (feasible).

Can we reach state with
$pc_0 = C_0$, $pc_1 = C_1$?

Answer: explore state space

forward, from initial state $(L_1, L_1, ?, ?, ?)$

is bad state reachable? or

backward, from error state $(C_0, C_1, ?, ?, ?)$

is initial state reachable?

A *model checker* implements traversal algorithms
also for more complex properties (*temporal logic*)

Model checking vs. graph traversal

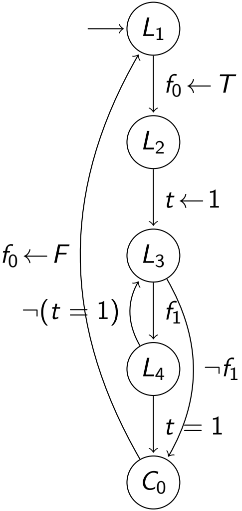
Simplest property: *reachability* – is error state reachable ?

We know this from graph traversal (BFS, DFS).
- but there, the graph is explicit and pre-build
- must only follow pointers from node to node

Model checking usually starts from a *model description* in text (program)
- C, Java, dedicated specification/modeling language

No pre-existing graph of nodes, model must be built
- e.g. explicit-state, on-the-fly state-space exploration
- or *symbolic*: state sets and transition relation are formulas
  - represented as *binary decision diagrams* (BDDs)
- may need to *compose* models (automata) for components

Everything is a formula



State sets are formulas over state variables:
$S_i = (pc_0 = 1) \wedge (pc_1 = 1)$ (initial)
f_0, f_1, t arbitrary $\Rightarrow$ 8 individual states

transition: formula over state and next state
$pc_0 = 1 \wedge pc_0' = 2 \wedge f_0' = 1$
$\wedge\ pc_1' = pc_1 \wedge t' = t \wedge f_1' = f_1$

Transition relation: disjunction ($\vee$) of all transitions

Next state set: all states s' such that
$s \in S_i \wedge step(s, s')$ i.e., $S_i(s) \wedge step(s, s')$

Finding an execution path

A path of length k from initial state set S_i to target state (set) S_f must satisfy

$$S_i(s_0) \wedge step(s_0, s_1) \wedge \ldots \wedge step(s_{k-1}, s_k) \wedge S_f(s_k)$$

This means *satisfiability checking* of a Boolean formula

NP-complete, but efficient algorithms in recent practice

Bounded model checking

If one can't explore the full state space, show that no error paths of length less than some k exist

Software model checking in practice

Early: SPIN tool (own modeling language with guarded commands)

SLAM project [Microsoft Research] (starting 2000)
(Software (Specifications), Languages, Analysis and Model checking)
later, many others: BLAST (UC Berkeley), CBMC (Oxford), ...
today: Software Verification Competition (5th edition, 2016)

Goal: checking *safety properties* (invariants)
  example: a program respects API usage rules
  calls to `lock()` and `unlock()` alternate

  used in practice for device drivers in Windows, Linux
  focused mostly on finding control/interface errors

Advantages:
– no need to annotate program by user
(only specify rules to monitor – simple automata)
– checking is automatic, for *all* possible executions
– generates *counterexample* (concrete execution) in case of error

Sample program

```
do {      // Device driver fragment [Ball & Rajamani '01]
  KeAcquireSpinLock(&devExt->writeListLock);
  nPacketsOld = nPackets;
  request = devExt->WriteListHeadVa;
  if(request && request->status) {
    devExt->WriteListHeadVa = request->Next;
    KeReleaseSpinLock(&devExt->writeListLock);
    irp = request->irp;
    if (request->status > 0) {
      irp->IoStatus.Status = STATUS_SUCCESS;
      irp->IoStatus.Information = request->Status;
    } else {
      irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
      irp->IoStatus.Information = request->Status;
    }
    SmartDevFreeBlock(request);
    IoCompleteRequest(irp, IO_NO_INCREMENT);
    nPackets++;
  }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock(&devExt->writeListLock);
```

Only highlighted code is relevant for correctness!

Specifying properties

A lock may be represented as one bit:
`acquire` and `release` change the bit value or signal error

```
state {
  enum { Unlocked=0, Locked=1 }
    state = Unlocked;
}
KeAcquireSpinLock.return {
  if (state == Locked) abort;
  else state = Locked;
}
KeReleaseSpinLock.return {
  if (state == Unlocked) abort;
  else state = Unlocked;
}
```

Given this lock model, the program is automatically instrumented
(original program is correct iff instrumented program can't reach error)

Abstraction is key to verification

Programs may be very complex
Many statements may be irrelevant for property of interest

$\Rightarrow$ want to focus on relevant program part

Program Slicing [Weiser, 1981]
determines program fragment (*slice*) that affects a given property (*slicing criterion*)
(e.g. value of a variable in a program point)

More generally: *abstraction*
generate a simplified program (model) from whose analysis we derive properties of the initial program

predicate = boolean condition (expression with program variables)

Generating the boolean program

Starts from the predicates in the specification

- nondeterministic branches
- `skip` (NOP) for irrelevant statements

Initially, keep just *control structure*, without data

```
do {
A: KeAcquireSpinLock_return();
  skip;
  if(*) {
B:   KeReleaseSpinLock_return();
    if (*) {
      skip;
    } else {
      skip;
    }
  }
} while (*);
C: KeReleaseSpinLock_return();
```

Model checking the boolean program

Abstract program is automaton: calculate reachable state set

state = program counter + variable assignment
state space: represented efficiently as boolean formula
(binary decision diagram, BDD)
computing with state sets: captures correlations between variables
transition relation: is also a boolean formula

$$state = 0 \wedge state' = 1$$

For given program, model checker finds error trace: may traverse
`A: KeAcquireSpinLock()` twice successively
if one never enters the `if` containing `B: Release...`

Is the error trace feasible ?

We get an error trace in the abstract program (model).
Is it feasible in the original (concrete) program ?

Map error trace onto original program
= find input values that satisfy constraints for the chosen path
(weakest preconditions)

If counterexample (error trace) is feasible, it is a real error.

If counterexample is not feasible, abstraction was too coarse
model myst be refined and re-checked
counterexample-guided abstraction refinement

Counterexample-guided abstraction refinement

In the given example, reproducing the counterexample fails
program exits `while` after first loop
⇒ the loop condition is *relevant* for the analyzed property

We introduce a new *predicate* (boolean variable) representing the condition

$$\texttt{b} \stackrel{\text{def}}{:=} \texttt{nPackets != nPacketsOld}$$

We generate a new boolean program ⇒ find statements depending on `b`.
Assignments `nPacketsOld = nPackets` and `nPackets++` *affect* `b`

We determine when after an assignment we know the value of `b` (true/false)
depending on all state bits (2^n for n predicates, here 1)

Abstracting statements

Find weakest precondition for `b`, resp. `!b` after given assignment.
We use for short `nP` and `nP0`.

We find *wp* for `b`: $\mathrm{wp}_T = \mathrm{wp}(\mathtt{nP} \leftarrow \mathtt{nP+1}, \mathtt{nP{=}nP0}) = \mathtt{nP+1{=}nP0}$
We check if `b` $\rightarrow \mathrm{wp}_T$ and if `!b` $\rightarrow \mathrm{wp}_T$
$\mathtt{nP{=}nP0} \not\rightarrow \mathtt{nP+1{=}nP0}$ and $\mathtt{nP{\neq}nP0} \not\rightarrow \mathtt{nP+1{=}nP0}$
So regardless of `b` we can't be sure that after `nP++`, `b` will be true.

We repeat with $\mathrm{wp}_F = \mathrm{wp}(\mathtt{nP} \leftarrow \mathtt{nP+1}, \mathtt{nP{\neq}nP0}) = \mathtt{nP+1{\neq}nP0}$
We have $\mathtt{nP{=}nP0} \rightarrow \mathtt{nP+1{\neq}nP0}$ and $\mathtt{nP{\neq}nP0} \not\rightarrow \mathtt{nP+1{\neq}nP0}$

So if `b` then after `nP++` we have `!b`, else we don't know.
$\Rightarrow$ we may abstract `nP++` with `b = b ? F : nondet`

Likewise, we may abstract `nP0 = nP` with `b = T`

Regenerate boolean program with the new predicates, check again.

Second boolean program

```
do {
A: KeAcquireSpinLock_return();
  b = T;     /* b == (nPackets == nPacketsOld) */
  if(*) {
B:   KeReleaseSpinLock_return();
    if (*) {
      skip;
    } else {
      skip;
    }
    b := choose(F, b);     // choose(p1, p2) == p1 ? T : p2 ? F : nondet
  }
} while (!b);
C: KeReleaseSpinLock_return();
```

Concluding...

The new abstraction is fine-grained enough.
Exploring all boolean program states the *model-checker* does not find an error path.

after `B:Release`, `b` becomes `F`, we stay in the cycle,
can't execute `C:Release` again (we do `A:Acquire`)

if we don't pass `B:Release`, `b` stays `T`, we exit the cycle,
can't repeat `A:Acquire` (we do `C:Release`)

May need several abstraction steps; termination not guaranteed.

In practice, *model checking* is feasible for *control*-rich programs: errors in drivers, Linux kernel, etc.