Model Checking Concurrent Software

Shaz Qadeer

Microsoft Research

Model checking, narrowly interpreted:

Decision procedures for checking if a given Kripke structure is a model for a given formula of a temporal logic.

Why is this of interest to us?

Because the dynamics of a discrete system can be captured by a Kripke structure.

Because some dynamic properties of a discrete system can be stated in temporal logics.

⇓

Model checking = System verification

Model checking, generously interpreted:

Algorithms, rather than proof calculi, for system verification which operate on a system model (semantics), rather than a system description (syntax).

A specific model-checking problem is defined by

I |= S

I: "implementation" (system model)

S: "specification" (system property)

|=: "satisfies", "implements", "refines" (satisfaction relation)

Paradigmatic example:

mutual-exclusion protocol

```
loop                                    ||  loop
  out:  x1 := 1; last := 1                    out:  x2 := 1; last := 2
  req:  await  x2 = 0  or  last = 2           req:  await  x1 = 0  or  last = 1
  in:   x1 := 0                               in:   x2 := 0
end loop.                                   end loop.

        P1                                          P2
```

Model-checking problem

I |= S

system model — system property — satisfaction relation

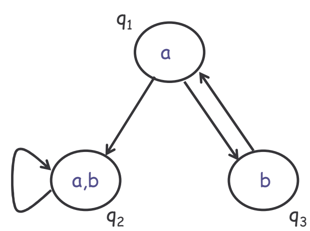
Model-checking problem

I |= S

system model — system property — satisfaction relation

While the choice of system model is important for ease of modeling in a given situation,

the only thing that is important for model checking is that the system model can be translated into some form of state-transition graph.



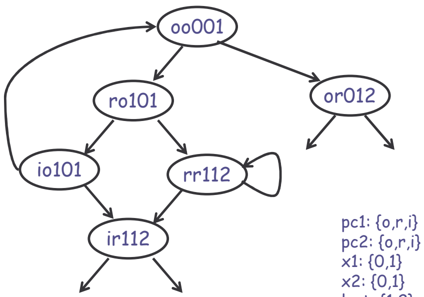
State-transition graph

Q	set of states	$\{q_1,q_2,q_3\}$
A	set of atomic observations	{a,b}
$\rightarrow \subseteq Q \times Q$	transition relation	$q_1 \rightarrow q_2$
$[\ ]: Q \rightarrow 2^A$	observation function	$[q_1] = \{a\}$

set of observations

Mutual-exclusion protocol

```
loop                                      ||  loop
  out:  x1 := 1; last := 1                      out:  x2 := 1; last := 2
  req:  await  x2 = 0  or  last = 2             req:  await  x1 = 0  or  last = 1
  in:    x1 := 0                                in:    x2 := 0
end loop.                                       end loop.

      P1                                             P2
```



pc1: {o,r,i}
pc2: {o,r,i}
x1: {0,1}
x2: {0,1}
last: {1,2}

$3 \cdot 3 \cdot 2 \cdot 2 \cdot 2 = 72$ states

The translation from a system description to a state-transition graph usually involves an exponential blow-up !!!

e.g., n boolean variables $\Rightarrow$ 2^n states

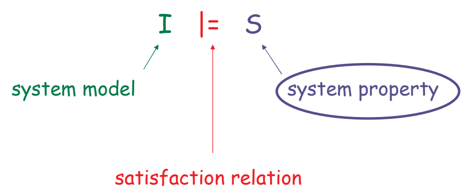
This is called the "state-explosion problem."

Finite state-transition graphs don't handle:
- recursion (need pushdown models)

State-transition graphs are not necessarily finite-state

We will talk about some of these issues later.

Model-checking problem



Example: Mutual exclusion

It cannot happen that both processes are in their critical sections simultaneously.

Initial states: $pc1 = o \wedge pc2 = o \wedge x1 = 0 \wedge x2 = 0$
Error states: $pc1 = r \wedge pc2 = r$

Reachability analysis:
Does there exist a path from an initial state to an error state?

Complexity of state transition graph is due to:

1. Control: finite (single program counter) vs.
   infinite (stack of program counters)
2. Data: finite domain (boolean) vs.
   infinite domain (integers) vs.
   dynamically created (heap objects)
3. Threads of control: single vs.
   multiple vs.
   dynamically created

For example, the mutual exclusion protocol has multiple threads of finite control and finite data.

Decidability of reachability analysis

Single thread of control:

Data \ Control	Finite: Acyclic	Finite: Looping	Infinite
Finite	Yes	Yes	Yes
Infinite	Yes	No	No

Decidability of reachability analysis

Multiple threads of control:

Data \ Control	Finite: Acyclic	Finite: Looping	Infinite
Finite	Yes	Yes	No
Infinite	Yes	No	No

Analysis of concurrent programs is difficult

- Finite-data finite control program
  - n lines
  - m states for global data variables
- 1 thread
  - n * m states
- K threads
  - $(n)^K * m$ states

Outline

- Reachability analysis for finite data
  - finite control
  - infinite control
- Richer property specifications
  - safety vs. liveness

Part 1: Reachability analysis for finite-state systems

Why should we bother about finite-data programs?

Two reasons:

1. These techniques are applicable to infinite-data programs without the guarantee of termination
2. These techniques are applicable to finite abstractions of infinite-data programs

Reachability analysis for finite data and finite control

1. Stateless model checking or systematic testing
   - enumerate executions
2. Explicit-state model checking with state caching
   - enumerate states

Note:
These techniques applicable even to infinite data and infinite control programs, but without the guarantee of termination.

Stateless model checking
a.k.a
Systematic testing

```
void doDfs() {
   stack.push(initialState);
   while (stack.Count > 0) {
    State s := (State) stack.Peek();

    // execute the next enabled thread
    int tid := s.NextEnabledThread();
    if (tid = -1) { stack.Pop(); continue; }
    State newS := s.Execute(tid);

    stack.push(newS);
}
```

This algorithm is not fully stateless since it requires a stack of states.

$$\text{init} \xrightarrow{t_1} \xrightarrow{t_2} \cdots \xrightarrow{t_n} s$$

Maintain instead a stack of thread identifiers.
To recreate the state at the top of the stack, replay the stack from the initial state

The algorithm will not terminate in general.
However, it will terminate if
- the program is acyclic
- if we impose a bound on the execution depth

Even if it terminates, it is very expensive
- after each step, every single thread is scheduled
- leads to too many executions

Atomic Increment

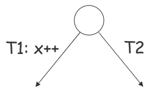
```
        int g = 0;

T1                  T2
int x = 0;          int y = 0;

x++;                y++;
g++;                g++;

x++;                y++;
g++;                g++;
```

Naïve stateless model checking:
No. of explored executions = $(4+4)!/(4!)^2 = 70$

No. of threads = n
No. of steps executed by each thread = k
No. of executions = (nk)! / (k!)^n

Partial-order reduction techniques

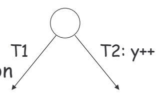
An access to x by T1 is invisible to T2.



Unnecessary to explore this transition

An access to y by T2 is invisible to T1.

Unnecessary to explore this transition



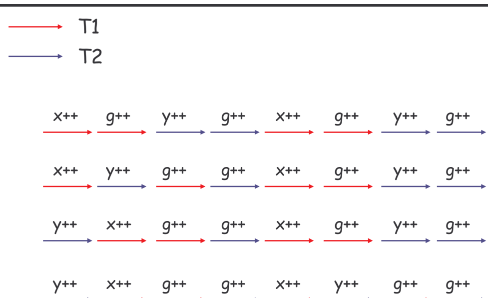
---

int g = 0;

T1	T2
int x = 0;	int y = 0;
x++;	y++;
g++;	g++;
x++;	y++;
g++;	g++;

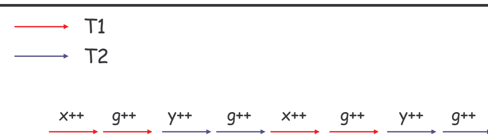
Without partial-order reduction:
No. of explored executions = $(4+4)!/(4!)^2 = 70$

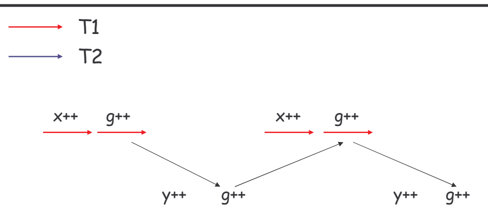
With partial-order reduction:
No. of explored executions = $(2+2)!/(2!)^2 = 6$



and so on …

Execution e1 is equivalent to e2 if e2 can be obtained from e1 by commuting adjacent independent operations.





An execution is partially rather than totally ordered!
- all linearizations of a partially-ordered execution are equivalent

Goal: an algorithm to systematically enumerate one and only one representative execution from each equivalence class

---

Lock l; int g = 0;

Non-atomic Increment

T1	T2
int x = 0;	int y = 0;
x++;	y++;
acq(l);	acq(l);
g++;	g++;
rel(l);	rel(l);
x++;	y++;
acq(l);	acq(l);
g++;	g++;
rel(l);	rel(l);

Challenge

Goal: an algorithm to systematically enumerate one and only one representative execution from each equivalence class

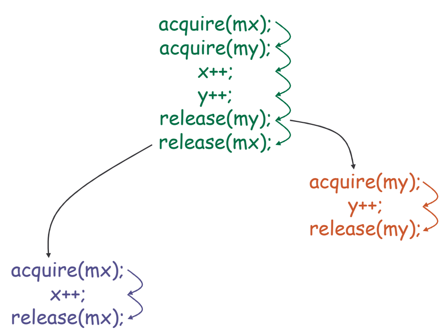
Obstacles:

1. Dependence between actions difficult to compute statically
2. Difficult to avoid repeating equivalent executions

Happens-before relation

- Partial-order on atomic actions in a concurrent execution
- Inter-thread edges based on program order
- Intra-thread edges based on synchronization actions
  - acquire and release on locks
  - fork and join on threads
  - P and V on semaphores
  - wait and signal on events

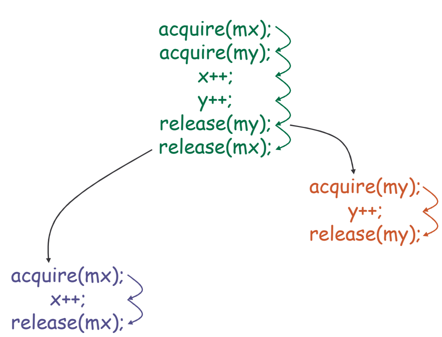
Happens-before relation



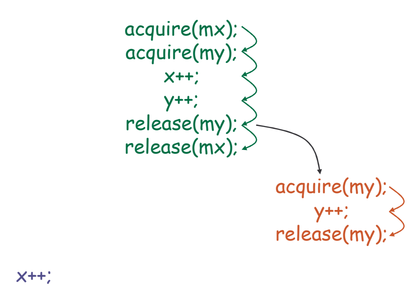
Data race

- Partition of program variables into synchronization variables and data variables
- There is a data-race on x if there are two accesses to x such that
  - They are unrelated by the happens-before relation
  - At least one of those accesses is a write

No race



Race on x



A data race usually indicates an error!

Improved partial-order reduction

- Schedule other threads only at accesses to synchronization variables
- Justified if each execution is free of data races
  - check by computing the happens-before relation
  - report each data race

Clock-vector algorithm

Initially:
Lock l: CV(l) = [0,…,0]   Thread t: CV(t) = [0,…,0]

Data variable x: Clock(x) = -1, Owner(x) = 0

Thread t performs:

release(l): CV(t)[t] := CV(t)[t] + 1; CV(l) := CV(t)

acquire(l): CV(t) := max(CV(t), CV(l))

```
access(x): if ( Owner(x) = t ∨ Clock(x) < CV(t)[Owner(x)] )
                   Owner(x) := t; Clock(x) := CV(t)[t]
           else
                   Report race on x
```

Further improvements

```
Lock lx, ly;   int x = 0, y = 0;

T1              T2

acq(lx);        acq(ly);
x++;            y++;
rel(lx);        rel(ly);
```

- Previous algorithm results in exploring two linearizations
- Yet, there is only one partially-ordered execution

Perform partial-order reduction on synchronization actions
- Flanagan-Godefroid 06
- Lei-Carver 06

Explicit-state model checking

- Explicitly generate the individual states
- Systematically explore the state space
  - State space: Graph that captures all behaviors
- Model checking = Graph search
- Generate the state space graph "on-the-fly"
  - State space is typically much larger than the reachable set of states

```
void doDfs() {
   while (stateStack.Count > 0) {
    State s := (State) stateStack.Peek();

    // execute the next enabled thread
    int tid := s.NextEnabledThread();
    if (tid = -1) { stateStack.Pop(); continue; }
    State newS := s.Execute(tid);

     if (stateHash.contains(newS)) continue;
     stateHash.add(newS);

     stateStack.push(newS);
}
```

State-space explosion

- Reachable set of states for realistic software is huge
- Need to investigate state-space reduction techniques
- Stack compression
- Identify behaviorally equivalent states
  - Process symmetry reduction
  - Heap symmetry reduction

Stack compression

- State vector can be very large
  - cloning the state vector to push an entry on the stack is expensive
- Each transition modifies only a small part of the state
- Solution
  - update state in place
  - push the state-delta on the stack

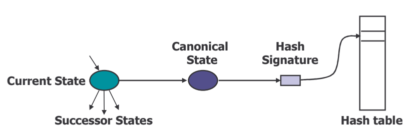
Hash compaction

- Compact states in the hash table [Stern, 1995]
  - Compute a signature for each state
  - Only store the signature in the hashtable
- Signature is computed incrementally
- Might miss errors due to collisions
- Orders of magnitude memory savings
  - Compact 100 kilobyte state to 4-8 bytes
- Possible to search ~10 million states

50

State symmetries

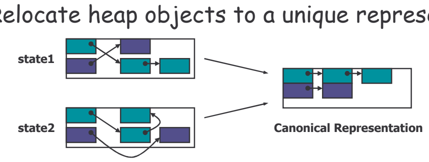
- Explore one out of a (large) set of equivalent states
- Canonicalize states before hashing



51

Heap canonicalization

- Heap objects can be allocated in different order
  - Depends on the order events happen
- Relocate heap objects to a unique representation



- Find a canonical representation for each heap graph by abstracting the concrete values of pointers

52

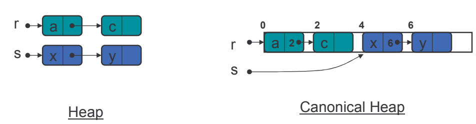
Heap-canonicalization algorithm

- Basic algorithm [Iosif 01]
  - Perform deterministic graph traversal of the heap (bfs / dfs)
  - Relocate objects in the order visited
- Incremental canonicalization [Musuvathi-Dill 04]
  - Should not traverse the entire heap in every transition

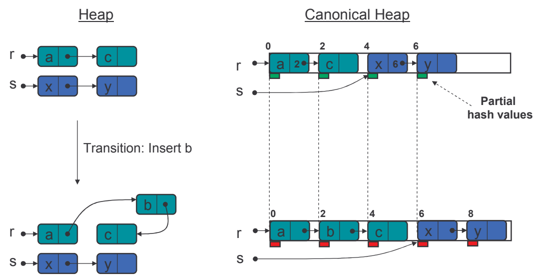
53

Iosif's canonicalization algorithm

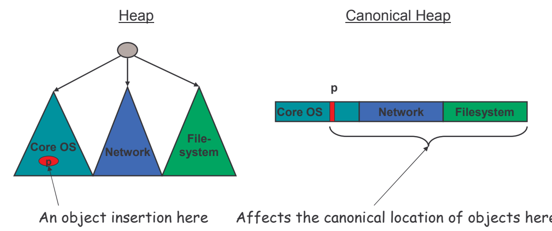
- Do a deterministic graph traversal of the heap (bfs / dfs)
- Relocate objects to a canonical location
  - Determined by the dfs (or bfs) number of the object
- Hash the resulting heap



Example: two linked lists

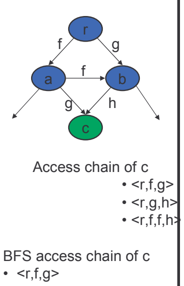


A Much Larger Example : Linux Kernel



Incremental heap canonicalization

- Access chain
  - A path from the root to an object in the heap
- BFS access chain
  - Shortest of all access paths from a global variable
  - Break ties lexicographically
- Canonical location of an object is a function of its bfs access chain



Revisiting example

Relocation Function Table				
	<r>	0	<s>	4
	<r,n>	2	<s,n>	6
	<r,n>	8		

r,s are root vars

n is the *next* field