System Specification, Verification and Synthesis (SSVS) – CS 4830/7485, Fall 2019

12: Formal Verification: Reachability

Stavros Tripakis

Northeastern University
Khoury College of Computer Sciences

Where we stand in the course

- Systems: DONE!
- Specification: Almost done! (we'll talk about automata later)
- Verification: next
- Synthesis: after that

Outline

- Verification
- Reachability analysis
- Counterexamples

VERIFICATION

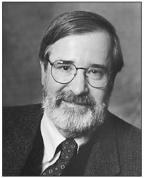
Verification and Computer-Aided Verification

- Systems: DONE!
- Specification: DONE (with temporal logics)!
- At this point, you should be able to do formal system modeling and specification.
- You could also in principle do verification "by hand", or using a general tool like a theorem-prover: plug in the definitions, try to prove the model-checking theorems.
- This is difficult to do by hand (theorem provers also typically require a lot of human interaction).
- So we turn to **computer-aided** and ideally **fully automated** verification.
- A.K.A. **model-checking**.

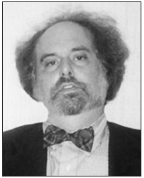
ACM Turing Award for Model-Checking

Clarke, Emerson, and Sifakis won the ACM Turing Award in 2007,

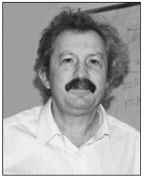
for their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries.



Edmund M. Clarke



E. Allen Emerson



Joseph Sifakis

Recall: the model-checking problems for LTL and CTL

Given:

- the implementation: a transition system (Kripke structure) $M = (\mathsf{AP}, S, S_0, L, R)$
- the specification: a temporal logic (LTL or CTL) formula ϕ

Check where M satisfies ϕ:

$$M \stackrel{?}{\models} \phi$$

- If ϕ is LTL: **every execution trace** of M must satisfy ϕ.
- If ϕ is CTL: **every initial state** of M must satisfy ϕ.

Recall: the model-checking problems for LTL and CTL

Given:

- the implementation: a transition system (Kripke structure) $M = (\mathsf{AP}, S, S_0, L, R)$
- the specification: a temporal logic (LTL or CTL) formula ϕ

Check where M satisfies ϕ:

$$M \overset{?}{\models} \phi$$

- If ϕ is LTL: **every execution trace** of M must satisfy ϕ.
- If ϕ is CTL: **every initial state** of M must satisfy ϕ.

For finite-state M, these questions can be answered fully automatically (problems are decidable)!

REACHABILITY ANALYSIS

Some model-checking problems are easier than others

For the same system M, some formulas may be easier to check than others.

Examples of two (conceptually) easy problems:

- checking deadlocks
- checking invariants

Checking Deadlock-Freedom

Checking that a system has no deadlocks (is **deadlock-free**) is conceptually easy:

- Explore (generate) all reachable states of the system.
- Check that none of them is a deadlock.[1]

[1] Some may be "legal end states", i.e., states without successors but which don't count as deadlocks because they have been identified (labeled) by the user as legal end states.

Recall: invariants

Suppose ϕ is of the form

$$\mathbf{G}\psi \qquad \text{or} \qquad \mathbf{AG}\psi$$

where ψ is a propositional formula (i.e., a boolean expression on atomic propositions).

E.g.,

$$\mathbf{G}(p \vee q), \qquad \mathbf{G}(p \rightarrow q), \qquad \cdots$$

Then ψ must be an **invariant**: it must hold at all reachable states.

Examples:

- "Whenever train is at intersection the gate must be lowered"
- "If the autopilot is off then the pilot must not believe it is on"

Model-Checking Invariants

Checking that ψ is an invariant is conceptually easy:

- Explore (generate) all reachable states.
- Check that every one of them satisfies ψ. (Is this easy? Why?)

Model-Checking Invariants

Checking that ψ is an invariant is conceptually easy:

- Explore (generate) all reachable states.
- Check that every one of them satisfies ψ. (Is this easy? Why?)

Caveat: this method is correct provided our system is deadlock-free. Why?

Model-Checking Invariants

Checking that ψ is an invariant is conceptually easy:

- Explore (generate) all reachable states.
- Check that every one of them satisfies ψ. (Is this easy? Why?)

Caveat: this method is correct provided our system is deadlock-free.
Why?
Only infinite paths count for the verification of a property such as $\mathbf{G}p$. If the system deadlocks after every time it violates p, then, formally speaking, it satisfies $\mathbf{G}p$!

So, what to do?

Model-Checking Invariants

Checking that ψ is an invariant is conceptually easy:

- Explore (generate) all reachable states.
- Check that every one of them satisfies ψ. (Is this easy? Why?)

Caveat: this method is correct provided our system is deadlock-free.
Why?
Only infinite paths count for the verification of a property such as $\mathbf{G}p$. If the system deadlocks after every time it violates p, then, formally speaking, it satisfies $\mathbf{G}p$!

So, what to do? Check deadlock-freedom before you check invariants!

They both use the same method: **reachability analysis**!

Reachability analysis

So, both for deadlocks and invariants, we want to:

- Explore (generate) all reachable states: this is called **reachability analysis**.

Sometimes it's also called **state-space exploration**.

Reachability analysis

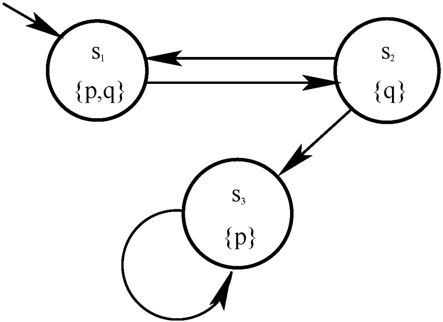
So, both for deadlocks and invariants, we want to:

- Explore (generate) all reachable states: this is called **reachability analysis**.

Sometimes it's also called **state-space exploration**.

- For finite-state systems, it can be done exhaustively and fully automatically!
- ... at least in theory ... in practice, often **state explosion** ...

Finite transition systems = Finite directed graphs



Any algorithm that explores all nodes of a graph can be used to explore all reachable states of a transition system!

Reachability analysis: summary

- Generate all reachable states ...
- ... while at the same time checking that each of them is "OK", i.e.,
    - it is not a deadlock state
    - it does not violate an invariant
    - ...

Reachability methods

- **Enumerative** (also called "explicit state").
  - These are basically search algorithms on directed graphs.

- **Symbolic** (we will see these later)
  - Bounded model-checking using SAT/SMT solvers.
  - Symbolic reachability.

ENUMERATIVE (EXPLICIT-STATE) REACHABILITY

Two standard search algorithms

- Depth-First Search (DFS)
- Breadth-First Search (BFS)

Depth-First Search

Assume given: Kripke structure (P, S, S_0, L, R).

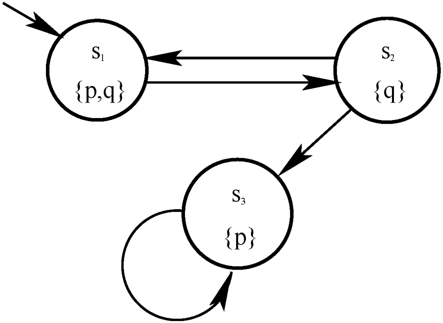
main:

```
V := ∅;                                      /* V: set of visited states */
for all s ∈ S0 do
    DFS(s);
end for
```

DFS(s):

```
check s;             /* is s a deadlock? is given p ∈ L(s)? ... */
V := V ∪ {s};
for all s′ such that (s, s′) ∈ R do
    if s′ ∉ V then
        DFS(s′);                             /* recursive call */
    end if
end for
```

Depth-First Search



Let's simulate DFS on this graph.

Depth-First Search

Quiz:

- Does DFS terminate?

Depth-First Search

Quiz:

- Does DFS terminate? Yes, if state space is finite.
- Does it visit all reachable states?

Depth-First Search

Quiz:

- Does DFS terminate? Yes, if state space is finite.
- Does it visit all reachable states? Yes: if s is reachable, then either $s \in S_0$, or s is the immediate successor of some s', which is itself reachable. In the first case, s is inserted into V because of the main loop. In the second case, assuming (by induction) that s' is inserted to V, s will also be inserted to V by loop in lines 3-6.
- Does it visit any unreachable states?

Depth-First Search

Quiz:

- Does DFS terminate? Yes, if state space is finite.
- Does it visit all reachable states? Yes: if s is reachable, then either $s \in S_0$, or s is the immediate successor of some s', which is itself reachable. In the first case, s is inserted into V because of the main loop. In the second case, assuming (by induction) that s' is inserted to V, s will also be inserted to V by loop in lines 3-6.
- Does it visit any unreachable states? No: following the "inverse" of the argument above, if s is inserted into V, either this is done because of the main loop, or because of the loop in lines 3-6. In the first case, s must be in S_0, so it's an initial state, so it's reachable. In the second case, s must be successor of some s', which by induction must be itself in V, therefore reachable.
- What is the complexity of the algorithm?

Depth-First Search

Quiz:

- Does DFS terminate? Yes, if state space is finite.
- Does it visit all reachable states? Yes: if s is reachable, then either $s \in S_0$, or s is the immediate successor of some s', which is itself reachable. In the first case, s is inserted into V because of the main loop. In the second case, assuming (by induction) that s' is inserted to V, s will also be inserted to V by loop in lines 3-6.
- Does it visit any unreachable states? No: following the "inverse" of the argument above, if s is inserted into V, either this is done because of the main loop, or because of the loop in lines 3-6. In the first case, s must be in S_0, so it's an initial state, so it's reachable. In the second case, s must be successor of some s', which by induction must be itself in V, therefore reachable.
- What is the complexity of the algorithm? $O(n + m)$ where n is number of nodes/states and m is number of edges/transitions in the graph. Every node and edge are visited at most once.

Breadth-First Search

Assume given: Kripke structure (P, S, S_0, L, R).

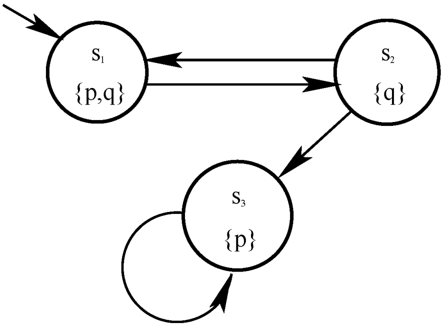
main:

FIFO queue $V := S_0$; /* V: queue of visited states */
set $E := \emptyset$; /* E: set of explored states */
BFS();

BFS:

while V non-empty **do**
  $s :=$ head(V);
  check s; /* is s a deadlock? is given $p \in L(s)$? ... */
  $E := E \cup \{s\}$;
  for all s' such that $(s, s') \in R$ and $s' \notin E \cup V$ **do**
    add s' to the end of queue V;
  end for
end while

Breadth-First Search



Let's simulate BFS on this graph.

Breadth-First Search

Quiz:

- Does BFS terminate?

Breadth-First Search

Quiz:

- Does BFS terminate? Yes, if state space is finite.
- Does it explore all reachable states?

Breadth-First Search

Quiz:

- Does BFS terminate? Yes, if state space is finite.
- Does it explore all reachable states? Yes: if s is reachable, then either $s \in S_0$, or s is the immediate successor of some s', which is itself reachable. In the first case, s is inserted initially into V. In the second case, assuming (by induction) that s' is inserted to V, s will also be inserted to V by loop in lines 5-7. All states in V are also eventually added in E.
- Does it explore any unreachable states?

Breadth-First Search

Quiz:

- Does BFS terminate? Yes, if state space is finite.
- Does it explore all reachable states? Yes: if s is reachable, then either $s \in S_0$, or s is the immediate successor of some s', which is itself reachable. In the first case, s is inserted initially into V. In the second case, assuming (by induction) that s' is inserted to V, s will also be inserted to V by loop in lines 5-7. All states in V are also eventually added in E.
- Does it explore any unreachable states? No: following the "inverse" of the argument above, if s is inserted into E, it must be first put in V. Either this is done initially, or because of the loop in lines 5-7. In the first case, s must be in S_0, so it's an initial state, so it's reachable. In the second case, s must be successor of some s', which by induction must be itself in V, therefore reachable.
- What is the complexity of the algorithm?

Breadth-First Search

Quiz:

- Does BFS terminate? Yes, if state space is finite.
- Does it explore all reachable states? Yes: if s is reachable, then either $s \in S_0$, or s is the immediate successor of some s', which is itself reachable. In the first case, s is inserted initially into V. In the second case, assuming (by induction) that s' is inserted to V, s will also be inserted to V by loop in lines 5-7. All states in V are also eventually added in E.
- Does it explore any unreachable states? No: following the "inverse" of the argument above, if s is inserted into E, it must be first put in V. Either this is done initially, or because of the loop in lines 5-7. In the first case, s must be in S_0, so it's an initial state, so it's reachable. In the second case, s must be successor of some s', which by induction must be itself in V, therefore reachable.
- What is the complexity of the algorithm? $O(n + m)$ where n is number of nodes/states and m is number of edges/transitions in the graph. Every node and edge are visited at most once.

Other enumerative algorithms

Every search algorithm on finite graphs can be used for reachability analysis:

- Best-first search:
  - every state is assigned a “value” (using some heuristic value function, e.g., how “close” we are likely to be to the goal – in our case a “bad” state) and then next state to explore is the one with the highest value.
- A*: classic search technique in artificial intelligence.
- ...

But isn't the complexity of graph search awesome?!

$O(m + n)$ is a great complexity, right?
Not really...

- Most of these algorithms (DFS, BFS, Best-first, A*, ...) have been tried by researchers in verification.
- Basic complexity is the same for all: need to store all reachable states
  - in the "worst case" from the algorithmic point of view
  - but in fact "best case" from the verification point of view, since we are trying to prove that our system is correct! $\Rightarrow$ all reachable states must be correct
- **State explosion**: the number of reachable states is too large 

The real complexity of reachability

Searching a graph is linear in the size of the graph, which appears to be a very nice worst-case complexity ...

The real complexity of reachability

Searching a graph is linear in the size of the graph, which appears to be a very nice worst-case complexity ...

... until we realize that the size of the graph is **exponential** in the number of state variables, processes, etc.

The real complexity of reachability

Searching a graph is linear in the size of the graph, which appears to be a very nice worst-case complexity ...

... until we realize that the size of the graph is **exponential** in the number of state variables, processes, etc.

This is not just a practical observation. There is theoretical complexity results about this, e.g., checking intersection emptiness of a **set** of DFA is PSPACE-complete.

The real complexity of reachability

Searching a graph is linear in the size of the graph, which appears to be a very nice worst-case complexity ...

... until we realize that the size of the graph is **exponential** in the number of state variables, processes, etc.

This is not just a practical observation. There is theoretical complexity results about this, e.g., checking intersection emptiness of a **set** of DFA is PSPACE-complete.

So even reachability is a hard problem (both theoretically and in practice).

Enumerative methods to remedy state explosion

- **Bit-state hashing**: instead of storing the entire state vector, just store 1 bit per state: its hash value [Holzmann, 1998].
  - Do you see a problem with this method?

Enumerative methods to remedy state explosion

- **Bit-state hashing**: instead of storing the entire state vector, just store 1 bit per state: its hash value [Holzmann, 1998].
  - Do you see a problem with this method?
  - **Incomplete**: two states may hash to the same value $\Rightarrow$ only one will be visited $\Rightarrow$ some reachable states may be missed!
  - And as we saw, even 1 bit per state may be too much already.

Enumerative methods to remedy state explosion

- **Bit-state hashing**: instead of storing the entire state vector, just store 1 bit per state: its hash value [Holzmann, 1998].
  - Do you see a problem with this method?
  - **Incomplete**: two states may hash to the same value $\Rightarrow$ only one will be visited $\Rightarrow$ some reachable states may be missed!
  - And as we saw, even 1 bit per state may be too much already.
- **Partial-order reduction**: in asynchronous concurrent systems, transitions of different processes are often independent $\Rightarrow$ no need to explore all interleavings [Valmari, 1990, Godefroid and Wolper, 1991].
- **Symmetry reduction**: many state spaces are symmetric $\Rightarrow$ equivalence relation on states $\Rightarrow$ suffices to explore just one state per equivalence class, e.g., see [Sistla and Godefroid, 2004].
- ...

All these help, but don't eliminate the state-explosion problem.
Note: above references are representative, there is a lot more work on these topics.

STATE EXPLOSION in Spin and nuXmv

State explosion in Spin

```
// an illustration of state explosion
// as you increase N, the state space increases exponentially

#define N 7

    active [N] proctype p()     // N processes
    {
        l0: skip;
        l1: skip;
        l2: skip;
        l3: skip;
        l4: skip;
        l5: skip;
        l6: skip;
        l7: skip;
    }
// analysis:
// spin -run -noreduce state-explosion.pml
// spin -run state-explosion.pml
```

Bibliography

Baier, C. and Katoen, J.-P. (2008).
Principles of Model Checking.
MIT Press.

Clarke, E., Grumberg, O., and Peled, D. (2000).
Model Checking.
MIT Press.

Godefroid, P. and Wolper, P. (1991).
Using partial orders for the efficient verification of deadlock freedom and safety properties.
In *4th CAV*.

Holzmann, G. (1998).
An analysis of bitstate hashing.
In *Formal Methods in System Design*, pages 301–314. Chapman & Hall.

Sistla, A. P. and Godefroid, P. (2004).
Symmetry and reduced symmetry in model checking.
ACM Trans. Program. Lang. Syst., 26(4):702–734.

Valmari, A. (1990).
Stubborn sets for reduced state space generation.
In *Advances in Petri Nets*, LNCS 483. Springer.