Model checking algorithms. Model checking in practice

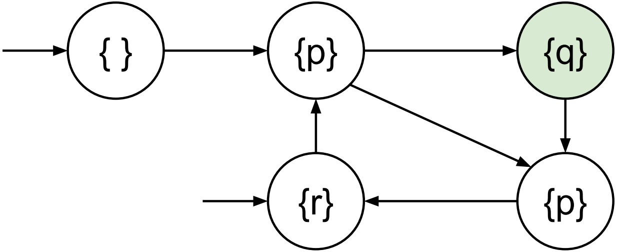
Igor Buzhinsky

igor.buzhinsky@gmail.com

August 19, 2020

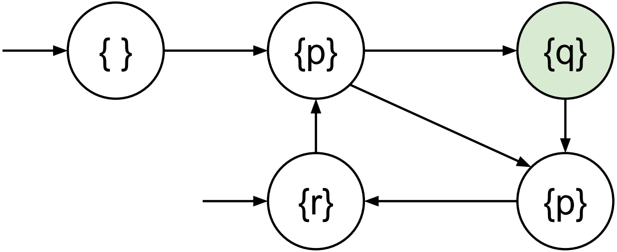
Explicit-space model checking algorithms

Explicit-state CTL model checking (1)



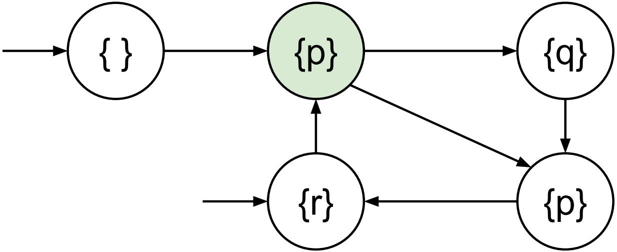
- How to model-check a Boolean formula (e.g., $f = q$, which is false here)?
- Just check it in all initial states

Explicit-state CTL model checking (1)



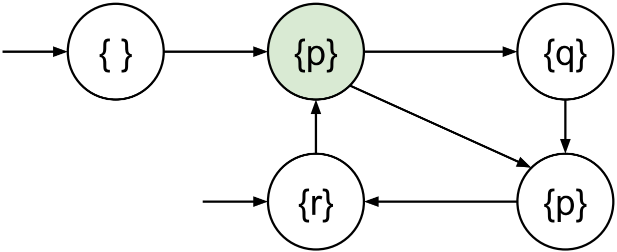
- How to model-check a Boolean formula (e.g., $f = q$, which is false here)?
- Just check it in all initial states
- The general algorithm will work as follows: *check it in all the states*, then report whether it is true in all initial states

Explicit-state CTL model checking (2)



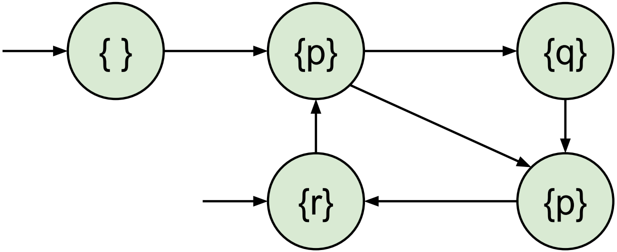
- Model checking $f = \mathbf{AX}g$ or $f = \mathbf{EX}g$, e.g., $f = \mathbf{AX}q$
- Find (recursively) the set of states S_g where g is satisfied
- $f = \mathbf{AX}g$: the answer is the set of states $S_f = \{b \in S \mid \forall b' : (b, b') \in T \; b' \in S_g\}$
- $f = \mathbf{EX}g$: the answer is the set of states $S_f = \{b \in S \mid \exists b' : (b, b') \in T \; b' \in S_g\}$

Explicit-state CTL model checking (2)



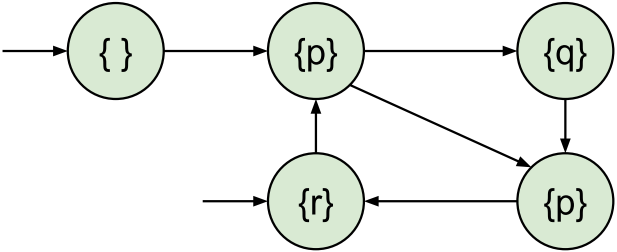
- Model checking $f = \mathbf{AX}g$ or $f = \mathbf{EX}g$, e.g., $f = \mathbf{AX}q$
- Find (recursively) the set of states S_g where g is satisfied
- $f = \mathbf{AX}g$: the answer is the set of states $S_f = \{b \in S \mid \forall b' : (b, b') \in T\ b' \in S_g\}$
- $f = \mathbf{EX}g$: the answer is the set of states $S_f = \{b \in S \mid \exists b' : (b, b') \in T\ b' \in S_g\}$
- Then (after all recursive calls) check that $I \subseteq S_f$

Explicit-state CTL model checking (3)



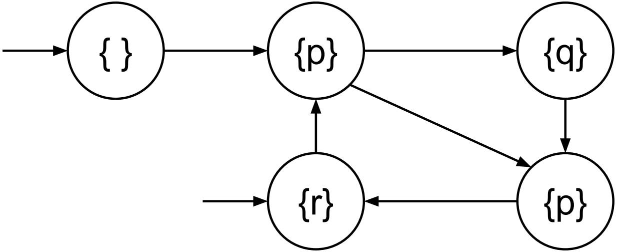
- Model checking $f = \mathbf{AF}g$ or $f = \mathbf{EF}g$, e.g., $f = \mathbf{EFAX}q$
- $f = \mathbf{EF}g$: $S_f = \{b \in S \mid \text{some path from } b \text{ eventually hits } S_g\}$
- $f = \mathbf{AF}g$: $S_f = \{b \in S \mid \text{all paths from } b \text{ eventually hit } S_g\}$

Explicit-state CTL model checking (3)



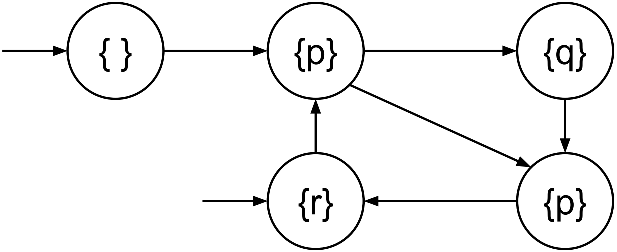
- Model checking $f = \mathbf{AF}g$ or $f = \mathbf{EF}g$, e.g., $f = \mathbf{EFAX}q$
- $f = \mathbf{EF}g$: $S_f = \{b \in S \mid \text{some path from } b \text{ eventually hits } S_g\}$
- $f = \mathbf{AF}g$: $S_f = \{b \in S \mid \text{all paths from } b \text{ eventually hit } S_g\}$
- Initialize S_f with S_g, then expand it with states from which some/all edges lead to the current S_f until further expansion becomes impossible (can be implemented with a FIFO queue)

Explicit-state CTL model checking (4)



- Model checking $f = \mathbf{AG}g$ or $f = \mathbf{EG}g$, e.g., $f = \mathbf{AGAX}q$
- $(s \models \mathbf{AG}g) \Leftrightarrow (s \models \neg\mathbf{EF}\neg g)$
- $(s \models \mathbf{EG}g) \Leftrightarrow (s \models \neg\mathbf{AF}\neg g)$

Explicit-state CTL model checking (4)



- Model checking $f = \mathbf{AG}g$ or $f = \mathbf{EG}g$, e.g., $f = \mathbf{AGAX}q$
- $(s \models \mathbf{AG}g) \Leftrightarrow (s \models \neg\mathbf{EF}\neg g)$
- $(s \models \mathbf{EG}g) \Leftrightarrow (s \models \neg\mathbf{AF}\neg g)$
- Apply the procedure from the previous slide to the complement of $S \setminus S_g$, then return the complement of the result

Explicit-state CTL model checking: summary

- Recursively, for each subformula of f, find the set of states where this subformula is satisfied
- Result $=$ whether all initial states belong to S_f

Explicit-state CTL model checking: summary

- Recursively, for each subformula of f, find the set of states where this subformula is satisfied
- Result = whether all initial states belong to S_f
- $O(|S|)$ time complexity for each operator
- $O(|S| \cdot |f|)$ time complexity for the entire formula f

Explicit-state CTL model checking: summary

- Recursively, for each subformula of f, find the set of states where this subformula is satisfied
- Result = whether all initial states belong to S_f
- $O(|S|)$ time complexity for each operator
- $O(|S| \cdot |f|)$ time complexity for the entire formula f
- Unfortunately, $O(|S|)$ is already expensive (state explosion problem)

Explicit-state CTL model checking: summary

- Recursively, for each subformula of f, find the set of states where this subformula is satisfied
- Result = whether all initial states belong to S_f
- $O(|S|)$ time complexity for each operator
- $O(|S| \cdot |f|)$ time complexity for the entire formula f
- Unfortunately, $O(|S|)$ is already expensive (state explosion problem)

- Question: how can the algorithm be modified to return counterexamples (e.g., for $f = \mathbf{AG}(p \to \mathbf{AX}q)$ or certificates (e.g., for $f = \mathbf{EFEX}p$)?

Explicit-state CTL model checking: summary

- Recursively, for each subformula of f, find the set of states where this subformula is satisfied
- Result = whether all initial states belong to S_f
- $O(|S|)$ time complexity for each operator
- $O(|S| \cdot |f|)$ time complexity for the entire formula f
- Unfortunately, $O(|S|)$ is already expensive (state explosion problem)

- Question: how can the algorithm be modified to return counterexamples (e.g., for $f = \mathbf{AG}(p \to \mathbf{AX}q)$ or certificates (e.g., for $f = \mathbf{EFEX}p$)?
- Question: how to model-check **AU** and **EU**?

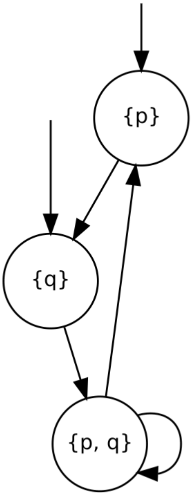
Explicit-state LTL model checking

- We wish to check whether f holds for Kripke structure M
- Automata-theoretic approach
- $\neg f$ is converted to a so-called **Büchi automaton**, which is an acceptor over infinite words that satisfy $\neg f$
- M is composed with this automaton
- If the composition accepts at least one infinite word, then this word satisfies $\neg f$ and belongs to M, so f is false, and the obtained word is a **counterexample**
- Otherwise, f is true

- We won't go into details

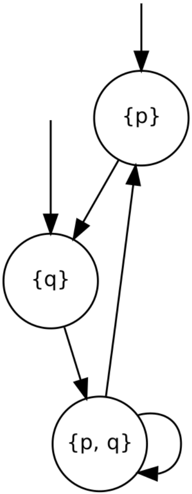
Symbolic model checking

State subsets as Boolean constraints (1)



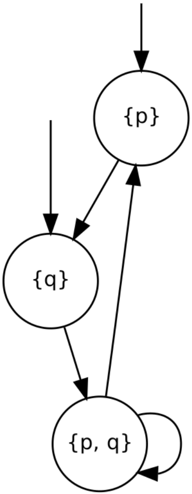
- Can you specify the set of reachable states as a Boolean formula?

State subsets as Boolean constraints (1)



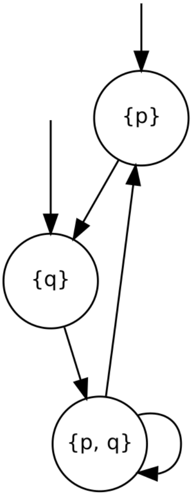
- Can you specify the set of reachable states as a Boolean formula?
- $p \vee q$

State subsets as Boolean constraints (1)



- Can you specify the set of reachable states as a Boolean formula?
- $p \vee q$
- What about only initial states?

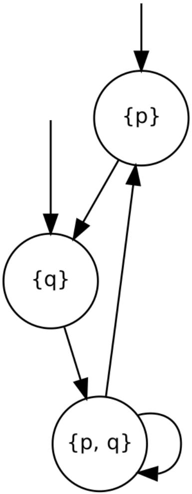
State subsets as Boolean constraints (1)



- Can you specify the set of reachable states as a Boolean formula?
- $p \vee q$

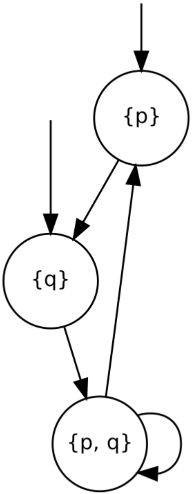
- What about only initial states?
- $p \oplus q = p \wedge \neg q \vee \neg p \wedge q$

State subsets as Boolean constraints (2)



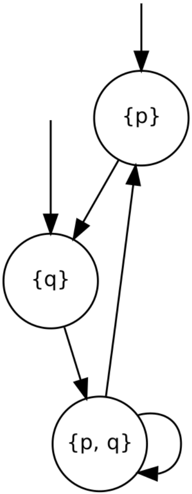
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step

State subsets as Boolean constraints (2)



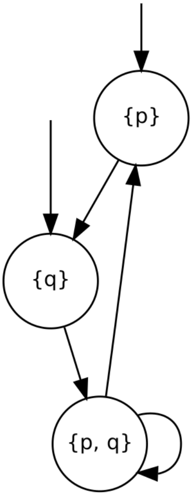
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula

State subsets as Boolean constraints (2)



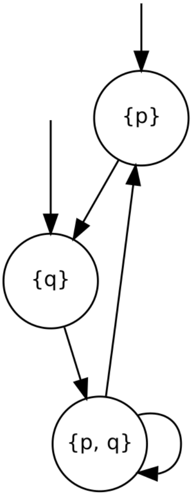
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
  - $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge$

State subsets as Boolean constraints (2)

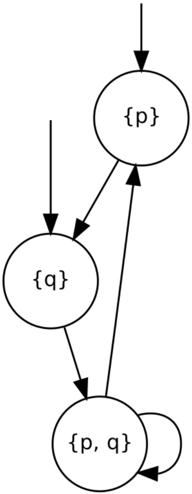


- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
  - $(p \land \neg q \rightarrow q' \land \neg p') \land (q \land \neg p \rightarrow p' \land q') \land$

State subsets as Boolean constraints (2)

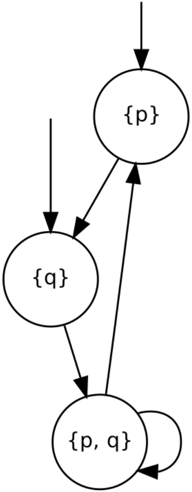


- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
  - $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge (q \wedge \neg p \rightarrow p' \wedge q') \wedge (p \wedge q \rightarrow p')$



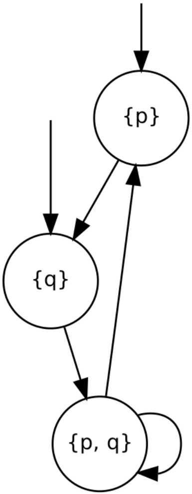
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
  - $(p \land \neg q \to q' \land \neg p') \land (q \land \neg p \to p' \land q') \land (p \land q \to p')$
  - Alternative way: $(p \land \neg q \land q' \land \neg p')$

State subsets as Boolean constraints (2)



- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
  - $(p \land \neg q \to q' \land \neg p') \land (q \land \neg p \to p' \land q') \land (p \land q \to p')$
  - Alternative way: $(p \land \neg q \land q' \land \neg p') \lor$

State subsets as Boolean constraints (2)



- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
  - $(p \land \neg q \to q' \land \neg p') \land (q \land \neg p \to p' \land q') \land (p \land q \to p')$
  - Alternative way: $(p \land \neg q \land q' \land \neg p') \lor (q \land \neg p \land p' \land q') \lor (p \land q \land p')$

Model checking with Boolean constraints?

- Assume that our Kripke structure has atomic propositions $p_1, \ldots, p_n$
- Boolean constraints $f_{\text{init}}[p_1, \ldots, p_n]$ and $f_{\text{trans}}[p_1, \ldots, p_n, p'_1, \ldots, p'_n]$
- How to model-check $g = \mathbf{AG}h$, where h is a Boolean formula?

Model checking with Boolean constraints?

- Assume that our Kripke structure has atomic propositions $p_1, \ldots, p_n$
- Boolean constraints $f_{\text{init}}[p_1, \ldots, p_n]$ and $f_{\text{trans}}[p_1, \ldots, p_n, p'_1, \ldots, p'_n]$
- How to model-check $g = \mathbf{AG}h$, where h is a Boolean formula?
- Compute a sequence of formulas f_i: the set of states reachable in i steps

Model checking with Boolean constraints?

- Assume that our Kripke structure has atomic propositions $p_1, \ldots, p_n$
- Boolean constraints $f_{\text{init}}[p_1, \ldots, p_n]$ and $f_{\text{trans}}[p_1, \ldots, p_n, p'_1, \ldots, p'_n]$
- How to model-check $g = \mathbf{AG}h$, where h is a Boolean formula?
- Compute a sequence of formulas f_i: the set of states reachable in i steps
- $f_0 := f_{\text{init}}$;

Model checking with Boolean constraints?

- Assume that our Kripke structure has atomic propositions $p_1, \ldots, p_n$
- Boolean constraints $f_{\text{init}}[p_1, \ldots, p_n]$ and $f_{\text{trans}}[p_1, \ldots, p_n, p'_1, \ldots, p'_n]$
- How to model-check $g = \mathbf{AG}h$, where h is a Boolean formula?
- Compute a sequence of formulas f_i: the set of states reachable in i steps
- $f_0 := f_{\text{init}}$; $f_i := f_{i-1} \vee \text{RemovePrimes}(\exists p_1, \ldots, p_n : f_{i-1} \wedge f_{\text{trans}})$

Model checking with Boolean constraints?

- Assume that our Kripke structure has atomic propositions $p_1, ..., p_n$
- Boolean constraints $f_{\mathrm{init}}[p_1, ..., p_n]$ and $f_{\mathrm{trans}}[p_1, ..., p_n, p'_1, ..., p'_n]$
- How to model-check $g = \mathbf{AG}h$, where h is a Boolean formula?
- Compute a sequence of formulas f_i: the set of states reachable in i steps
- $f_0 := f_{\mathrm{init}}$; $f_i := f_{i-1} \vee \mathrm{RemovePrimes}(\exists p_1, ..., p_n : f_{i-1} \wedge f_{\mathrm{trans}})$
- If $f_i \wedge \neg h$ is **satisfiable**, then g is false

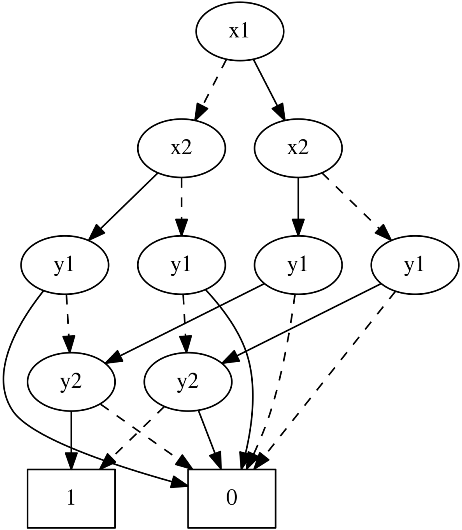
Model checking with Boolean constraints?

- Assume that our Kripke structure has atomic propositions $p_1, \ldots, p_n$
- Boolean constraints $f_{\text{init}}[p_1, \ldots, p_n]$ and $f_{\text{trans}}[p_1, \ldots, p_n, p'_1, \ldots, p'_n]$
- How to model-check $g = \mathbf{AG}h$, where h is a Boolean formula?
- Compute a sequence of formulas f_i: the set of states reachable in i steps
- $f_0 := f_{\text{init}}$; $f_i := f_{i-1} \vee \text{RemovePrimes}(\exists p_1, \ldots, p_n : f_{i-1} \wedge f_{\text{trans}})$
- If $f_i \wedge \neg h$ is **satisfiable**, then g is false
- If at some point f_i and f_{i-1} become **equivalent**, we can stop the procedure and conclude that g is true

Model checking with Boolean constraints?

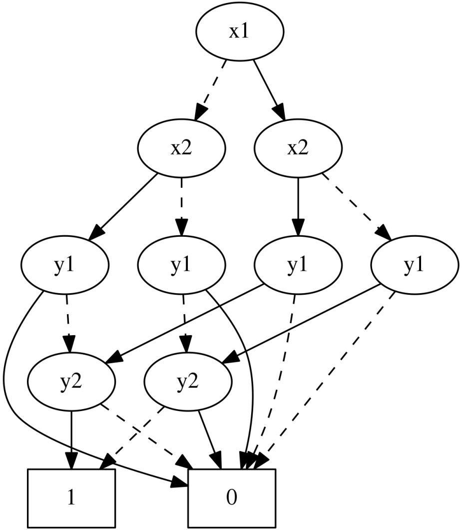
- Assume that our Kripke structure has atomic propositions $p_1, \ldots, p_n$
- Boolean constraints $f_{\text{init}}[p_1, \ldots, p_n]$ and $f_{\text{trans}}[p_1, \ldots, p_n, p'_1, \ldots, p'_n]$
- How to model-check $g = \mathbf{AG}h$, where h is a Boolean formula?
- Compute a sequence of formulas f_i: the set of states reachable in i steps
- $f_0 := f_{\text{init}}$; $f_i := f_{i-1} \vee \text{RemovePrimes}(\exists p_1, \ldots, p_n : f_{i-1} \wedge f_{\text{trans}})$
- If $f_i \wedge \neg h$ is **satisfiable**, then g is false
- If at some point f_i and f_{i-1} become **equivalent**, we can stop the procedure and conclude that g is true
- How to perform all these symbolic operations efficiently? There are binary decision diagrams (BDDs), a reduced form of decision trees

Example of a BDD



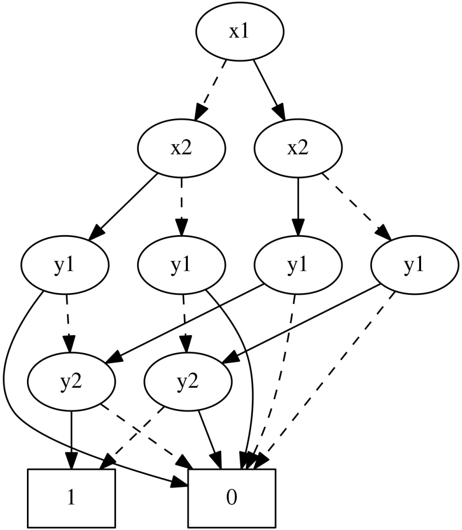
- Solid arrows: variable is true
- Dashed arrows: variable is false
- If in the end we come to 1, then the formula is true for our assignment
- If we come to 0, it is false

Example of a BDD



- Solid arrows: variable is true
- Dashed arrows: variable is false
- If in the end we come to 1, then the formula is true for our assignment
- If we come to 0, it is false
- Which function is encoded in this BDD?

Example of a BDD



- Solid arrows: variable is true
- Dashed arrows: variable is false
- If in the end we come to 1, then the formula is true for our assignment
- If we come to 0, it is false
- Which function is encoded in this BDD?
  – $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$

Literature

- Baier, C., & Katoen, J. P. (2008). Principles of model checking. MIT press
- Clarke, E. M., Grumberg, O., & Peled, D. (1999). Model checking. MIT press
- Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., & Hwang, L. J. (1992). Symbolic model checking: 10^{20} states and beyond. Information and computation, 98(2), pp. 142–170

Challenges of model checking

Why is it difficult to apply model checking in industry?

Why is it difficult to apply model checking in industry?

Computational difficulty

- **State space explosion:** in explicit-state model checkers, verification time and required RAM generally grows linearly with the growth of the state space

Why is it difficult to apply model checking in industry?

Computational difficulty

- **State space explosion:** in explicit-state model checkers, verification time and required RAM generally grows linearly with the growth of the state space
- **Model complexity** can be problematic even for symbolic model checkers

Why is it difficult to apply model checking in industry?

Computational difficulty

- **State space explosion:** in explicit-state model checkers, verification time and required RAM generally grows linearly with the growth of the state space
- **Model complexity** can be problematic even for symbolic model checkers

Difficulty related to human resources

- Model checking algorithms are automated, but the rest is often not

Why is it difficult to apply model checking in industry?

Computational difficulty

- **State space explosion:** in explicit-state model checkers, verification time and required RAM generally grows linearly with the growth of the state space
- **Model complexity** can be problematic even for symbolic model checkers

Difficulty related to human resources

- Model checking algorithms are automated, but the rest is often not
- Need to **prepare formal models**. Often, this involves finding a trade-off between computational complexity and precision of modeling

Why is it difficult to apply model checking in industry?

Computational difficulty

- **State space explosion:** in explicit-state model checkers, verification time and required RAM generally grows linearly with the growth of the state space
- **Model complexity** can be problematic even for symbolic model checkers

Difficulty related to human resources

- Model checking algorithms are automated, but the rest is often not
- Need to **prepare formal models**. Often, this involves finding a trade-off between computational complexity and precision of modeling
- Need to **prepare formal specifications**. Often, from natural language documents

Why is it difficult to apply model checking in industry?

Computational difficulty

- **State space explosion:** in explicit-state model checkers, verification time and required RAM generally grows linearly with the growth of the state space
- **Model complexity** can be problematic even for symbolic model checkers

Difficulty related to human resources

- Model checking algorithms are automated, but the rest is often not
- Need to **prepare formal models**. Often, this involves finding a trade-off between computational complexity and precision of modeling
- Need to **prepare formal specifications**. Often, from natural language documents
- **Knowledge and experience** are required to use formal methods correctly and efficiently

Why is it difficult to apply model checking in industry?

Computational difficulty

- **State space explosion:** in explicit-state model checkers, verification time and required RAM generally grows linearly with the growth of the state space
- **Model complexity** can be problematic even for symbolic model checkers

Difficulty related to human resources

- Model checking algorithms are automated, but the rest is often not
- Need to **prepare formal models**. Often, this involves finding a trade-off between computational complexity and precision of modeling
- Need to **prepare formal specifications**. Often, from natural language documents
- **Knowledge and experience** are required to use formal methods correctly and efficiently
- **Human factor**

User-friendly specification representation and editing

Patterns by Dwyer et al. (1998, 1999): example 1, **Absence** pattern

Absence

Intent

To describe a portion of a system's execution that is free of certain events or states. Also known as **Never**.

Example Mappings

CTL P is false:

Globally	$AG(\neg P)$
Before R	$A[\neg P\, \mathcal{U} (R \vee AG(\neg R))]$
After Q	$AG(Q \to AG(\neg P))$
Between Q and R	$AG(Q \to A[\neg P\, \mathcal{U} (R \vee AG(\neg R))])$
After Q until R	$AG(Q \to \neg E[\neg R\, \mathcal{U} (P \wedge \neg R)])$

LTL P is false:

Globally	$\Box(\neg P)$
Before R	$\Diamond R \to \neg P\, \mathcal{U}\, R$
After Q	$\Box(Q \to \Box(\neg P))$
Between Q and R	$\Box((Q \wedge \circ\Diamond R) \to (\neg P \wedge \circ(\neg P\, \mathcal{U}\, R)))$
After Q until R	$\Box(Q \to (\neg P \wedge \circ(\neg P\, \mathcal{U} (R \vee \Box\neg P))))$

- "A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system"

Patterns by Dwyer et al. (1998, 1999): example 2, **Response** pattern

Response

Intent

To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect, within a defined portion of a system's execution. Also known as **Follows** and **Leads-to**.

Example Mappings

In these mappings P is the cause and S is the effect.

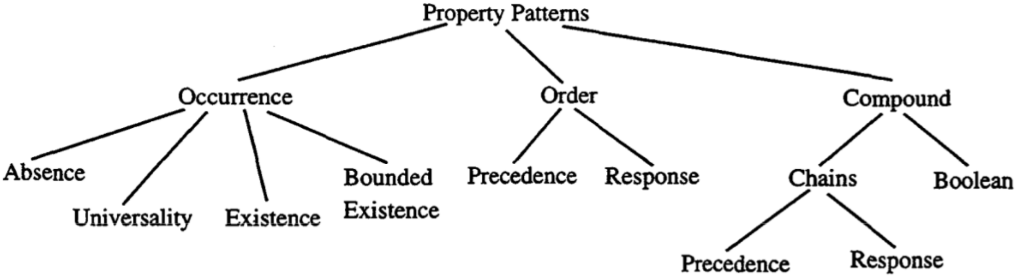
CTL S responds to P:

Globally	$AG(P \to AF(S))$
Before R	$A[(P \to A[\neg R\, \mathcal{U}((S \wedge \neg R) \vee AG(\neg R))])\, \mathcal{U}(R \vee AG(\neg R))]$
After Q	$AG(Q \to AG(P \to AF(S)))$
Between Q and R	$AG(Q \to A[(P \to A[\neg R\, \mathcal{U}((S \wedge \neg R) \vee AG(\neg R))])\, \mathcal{U}(R \vee AG(\neg R))])$
After Q until R	$AG(Q \to \neg E[\neg R\, \mathcal{U} \neg(P \to A[\neg R\, \mathcal{U}\, S]) \wedge \neg R])$

LTL S responds to P:

Globally	$\Box(P \to \Diamond S)$
Before R	$(P \to (\neg R\, \mathcal{U}(S \wedge \neg R)))\, \mathcal{U}(R \vee \Box\neg R)$
After Q	$\Box(Q \to \Box(P \to \Diamond S))$
Between Q and R	$\Box((Q \wedge \circ\Diamond R) \to (P \to (\neg R\, \mathcal{U}(S \wedge \neg R)))\, \mathcal{U}\, R)$
After Q until R	$\Box(Q \to ((P \to (\neg R\, \mathcal{U}(S \wedge \neg R)))\, \mathcal{U}\, R) \vee \Box(P \to (\neg R\, \mathcal{U}(S \wedge \neg R))))$

Patterns by Dwyer et al. (1998, 1999): hierarchy



- These patterns were extracted based on a volume of temporal properties collected from literature, student projects and other researchers
- Note: different domains may have different prevailing patterns

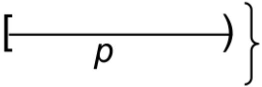
Visual specification languages (VSLs)

- Techniques to allow property representation and editing in a user-friendly, visual way
- Ideally, such techniques must be supported by tools
- Ideally, such tools must automatically translate visual specifications to textual formal specification languages (e.g. LTL, CTL)
- Unfortunately, this is not always so

- On the following slides, several examples of VSLs are shown, but more exist

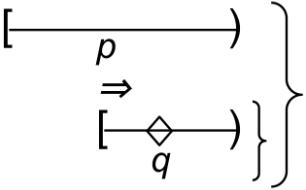
Visual specification languages: graphical interval logic

“Always” $\mathbf{G}\, p$



The single interval corresponds to the entire system run, during which p is universally asserted by drawing it below the center of the interval.

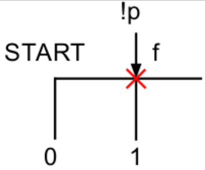
“Unbounded response” $\mathbf{G}(p \to \mathbf{F}\, q)$



For each point of the entire system run, the infinite time interval starting from this point is considered. If p is true at this point, then q is asserted somewhere in the future by drawing it below the diamond.

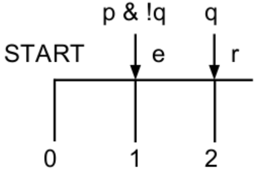
Visual specification languages: TimeLine editor

"Always"	$\mathbf{G}\, p$



$\mathbf{G}\, p$ is asserted by prohibiting $\neg p$ (using a fail event). There are no regular events, so this prohibition is valid universally.

"Unbounded response"	$\mathbf{G}(p \to \mathbf{F}\, q)$



If $p \wedge \neg q$ is true (regular event), then q must happen in the future (required event).

Literature

- Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1998). Property specification patterns for finite-state verification. Second workshop on Formal methods in software practice, pp. 7–15. ACM
- Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. International Conference on Software Engineering, pp. 411–420. IEEE
- Pang, C., Pakonen, A., Buzhinsky, I., & Vyatkin V. A study on user-friendly formal specification languages for requirements formalization. (2016). IEEE International Conference on Industrial Informatics (INDIN 2016), pp. 676–682
- Pakonen A., Pang C., Buzhinsky I., Vyatkin V. User-friendly formal specification languages – conclusions drawn from industrial experience on model checking. IEEE International Conference on Emerging Technologies & Factory Automation (ETFA 2016)

Explanation and visualization of counterexamples

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model
- In general, these sequences are infinite (prefix + loop)

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model
- In general, these sequences are infinite (prefix + loop)
  - Question: when a finite prefix is enough?

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model
- In general, these sequences are infinite (prefix + loop)
  - Question: when a finite prefix is enough? – for example, to show the falsity of $\mathbf{G}\neg x$, it is sufficient to provide a path up to the moment when x is reached

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model
- In general, these sequences are infinite (prefix + loop)
  - Question: when a finite prefix is enough? – for example, to show the falsity of $\mathbf{G}\neg x$, it is sufficient to provide a path up to the moment when x is reached
  - Question: are there cases when the temporal formula is violated but there are no counterexamples?

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model
- In general, these sequences are infinite (prefix + loop)
    - Question: when a finite prefix is enough? – for example, to show the falsity of $\mathbf{G}\neg x$, it is sufficient to provide a path up to the moment when x is reached
    - Question: are there cases when the temporal formula is violated but there are no counterexamples? – for some CTL properties, such as reachability ones (e.g. $\mathbf{EF}x$, $\mathbf{AGEF}x$)

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model
- In general, these sequences are infinite (prefix + loop)
  - Question: when a finite prefix is enough? – for example, to show the falsity of $\mathbf{G}\neg x$, it is sufficient to provide a path up to the moment when x is reached
  - Question: are there cases when the temporal formula is violated but there are no counterexamples? – for some CTL properties, such as reachability ones (e.g. $\mathbf{EF}x$, $\mathbf{AGEF}x$)
- Any problems with such a value table?

Counterexample explanation

- Counterexamples are intended to explain why the temporal property is violated
- They are merely sequences of values of variables in the model
- In general, these sequences are infinite (prefix + loop)
  - Question: when a finite prefix is enough? – for example, to show the falsity of $\mathbf{G}\neg x$, it is sufficient to provide a path up to the moment when x is reached
  - Question: are there cases when the temporal formula is violated but there are no counterexamples? – for some CTL properties, such as reachability ones (e.g. $\mathbf{EF}x$, $\mathbf{AGEF}x$)
- Any problems with such a value table? – the analysis may be troublesome due to a large number of variables and/or time instants

Example of a NuSMV counterexample

```
-- specification G(((!alarm & !criteria) & X (criteria & !ack_button)) -> X alarm)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    ack_button = TRUE
    alarm = FALSE
    criteria = TRUE
    ack_button_FAULT = FALSE
    criteria_FAULT = FALSE
    alarm_FAULT = FALSE
  -> State: 1.2 <-
    criteria = FALSE
  -- Loop starts here
  -> State: 1.3 <-
    ack_button = FALSE
    criteria = TRUE
  -> State: 1.4 <-
    ack_button = TRUE
  -> State: 1.5 <-
    ack_button = FALSE
```

Why is the formula violated?

```
-> State: 1.1 <-
  ack_button = TRUE
  alarm = FALSE
  criteria = TRUE
  ack_button_FAULT = FALSE
  criteria_FAULT = FALSE
  alarm_FAULT = FALSE
-> State: 1.2 <-
  criteria = FALSE
-- Loop starts here
-> State: 1.3 <-
  ack_button = FALSE
  criteria = TRUE
-> State: 1.4 <-
  ack_button = TRUE
-> State: 1.5 <-
  ack_button = FALSE
```

- $f = \mathbf{G}(((\neg\mathtt{alarm} \land \neg\mathtt{criteria}) \land \mathbf{X}(\mathtt{criteria} \land \neg\mathtt{ack\_button})) \rightarrow \mathbf{X}\mathtt{alarm})$
- Remember that NuSMV only reports new variable values, e.g. `alarm` is actually false everywhere
- **Structural approach**: (1) precompute the values of all subformulas on all time instants and (2) explain the violation recursively, starting from the **top-most operator** and the **first time instant**

Why is the formula violated?

```
-> State: 1.1 <-
  ack_button = TRUE
  alarm = FALSE
  criteria = TRUE
  ack_button_FAULT = FALSE
  criteria_FAULT = FALSE
  alarm_FAULT = FALSE
-> State: 1.2 <-
  criteria = FALSE
-- Loop starts here
-> State: 1.3 <-
  ack_button = FALSE
  criteria = TRUE
-> State: 1.4 <-
  ack_button = TRUE
-> State: 1.5 <-
  ack_button = FALSE
```

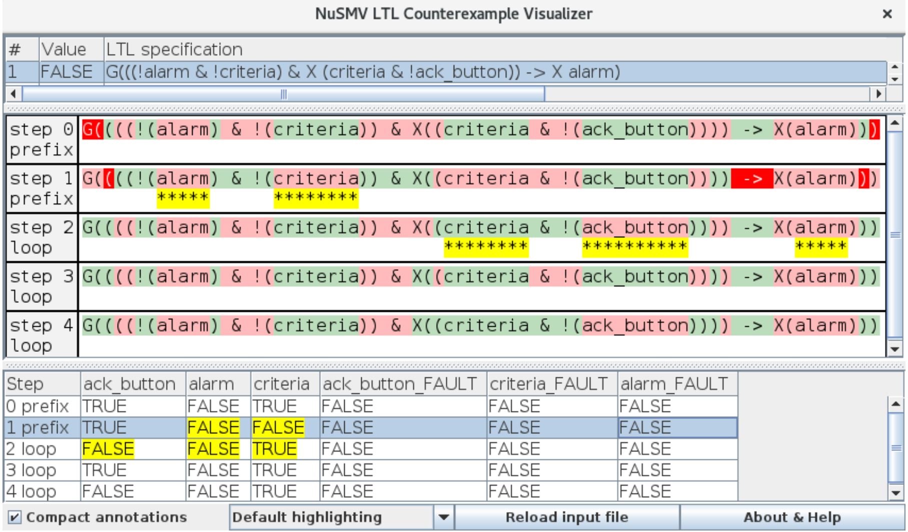
- $f = \mathbf{G}(((\neg\texttt{alarm} \land \neg\texttt{criteria}) \land \mathbf{X}(\texttt{criteria} \land \neg\texttt{ack\_button})) \to \mathbf{X}\texttt{alarm})$
- Remember that NuSMV only reports new variable values, e.g. `alarm` is actually false everywhere
- **Structural approach**: (1) precompute the values of all subformulas on all time instants and (2) explain the violation recursively, starting from the **top-most operator** and the **first time instant**
- Why $\mathbf{G}(\ldots)$ is false in state 1.1? Let's see in which states its argument is false!

Why is the formula violated?

```
-> State: 1.1 <-
  ack_button = TRUE
  alarm = FALSE
  criteria = TRUE
  ack_button_FAULT = FALSE
  criteria_FAULT = FALSE
  alarm_FAULT = FALSE
-> State: 1.2 <-
  criteria = FALSE
-- Loop starts here
-> State: 1.3 <-
  ack_button = FALSE
  criteria = TRUE
-> State: 1.4 <-
  ack_button = TRUE
-> State: 1.5 <-
  ack_button = FALSE
```

- $f = \mathbf{G}(((\neg\texttt{alarm} \land \neg\texttt{criteria}) \land \mathbf{X}(\texttt{criteria} \land \neg\texttt{ack\_button})) \to \mathbf{X}\texttt{alarm})$
- Remember that NuSMV only reports new variable values, e.g. `alarm` is actually false everywhere
- **Structural approach**: (1) precompute the values of all subformulas on all time instants and (2) explain the violation recursively, starting from the **top-most operator** and the **first time instant**
- Why **G**(...) is false in state 1.1? Let's see in which states its argument is false! – in state 1.2

This analysis can be automated!



About the tool

- This is the same tool as the one shown on the previous lecture
- Download from
  `https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer`
- The tool implements the approach proposed by Beer et al. (2012)

- Clicking on an operator will highlight the immediate causes of the value of the corresponding subformula

About the tool

- This is the same tool as the one shown on the previous lecture
- Download from `https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer`
- The tool implements the approach proposed by Beer et al. (2012)

- Clicking on an operator will highlight the immediate causes of the value of the corresponding subformula
- Important variable values are highlighted
- How to obtain them automatically?

About the tool

- This is the same tool as the one shown on the previous lecture
- Download from `https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer`
- The tool implements the approach proposed by Beer et al. (2012)

- Clicking on an operator will highlight the immediate causes of the value of the corresponding subformula
- Important variable values are highlighted
- How to obtain them automatically? – run the recursion to the end; the leaves of the tree will correspond to these values
- What is the meaning of highlighting?

About the tool

- This is the same tool as the one shown on the previous lecture
- Download from `https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer`
- The tool implements the approach proposed by Beer et al. (2012)

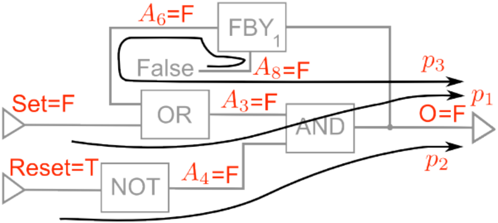
- Clicking on an operator will highlight the immediate causes of the value of the corresponding subformula
- Important variable values are highlighted
- How to obtain them automatically? – run the recursion to the end; the leaves of the tree will correspond to these values
- What is the meaning of highlighting? – the highlighted values are **sufficient** to **cause** the overall false outcome
- If you are interested, in (Beer et al. 2012) there is also a definition of causality

Counterexample explanation using paths in the function block diagram

- Often some variables may be relevant but are not included into the temporal formula
- Thus, the approach considered on the previous slides can do nothing with them!
- In (Bochot et al. 2010), an approach was proposed to explain particular values using paths in the diagram of modules

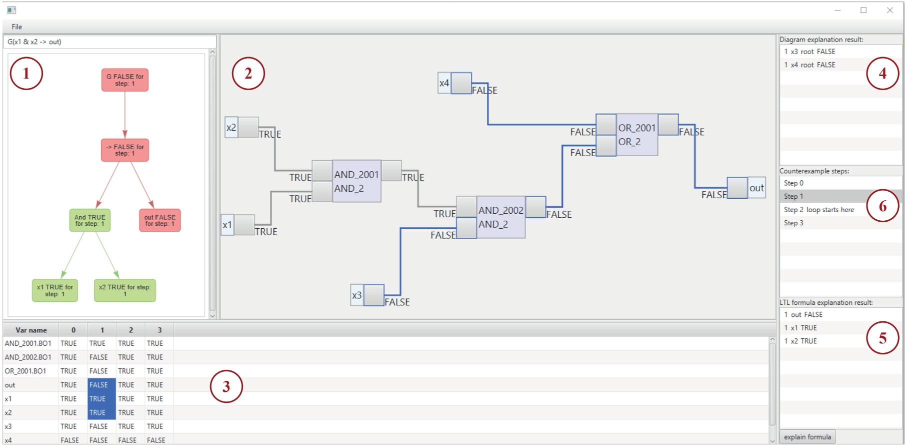
Counterexample explanation using paths in the function block diagram

- Often some variables may be relevant but are not included into the temporal formula
- Thus, the approach considered on the previous slides can do nothing with them!
- In (Bochot et al. 2010), an approach was proposed to explain particular values using paths in the diagram of modules
- Consider the figure on the right and a counterexample consisting of one cycle



- FBY_1 is a one-cycle delay (it is just initialized with False)
- The false value of O is explained with paths $\{p_2\}$, or with paths $\{p_1, p_3\}$

One more user-friendly tool: Oeritte



- Finding causes both in the temporal formula and **in the diagram of NuSMV modules**
- `https://github.com/ShakeAnApple/cxbacktracker`

Literature

- Beer, I., Ben-David S., Chockler H., Orni A., Trefler R. Explaining counterexamples using causality. Formal Methods in System Design, vol. 40, no. 1, pp. 20–40, 2012
- Pakonen A., Buzhinsky I., Vyatkin V. Counterexample visualization and explanation for function block diagrams. 16th IEEE International Conference on Industrial Informatics (INDIN 2018), pp. 747–753
- Bochot T., Virelizier P., Waeselynck H., Wiels V. Paths to property violation: A structural approach for analyzing counterxamples. IEEE 12th International Symposium on High Assurance Systems Engineering, pp. 74–83, 2010

Explicit-state vs. symbolic model checking in terms of computational complexity

Symbolic vs. explicit-state model checking

- Explicit-state model checking: the state space of the system is stored and analyzed explicitly, as a directed graph
- Time and memory complexity of explicit-state model checking is linear with respect to the size of the state space

- Symbolic model checking: operations with the state space are performed implicitly since its subsets can be represented as Boolean formulas
- Binary decision diagrams (BDDs) allow efficient operations with them
- The complexity of symbolic model checking does not explicitly depend on the number of states

Instability of BDD-based model checking time

- Example: elevator model parameterized by the number of floors n
- The state space grows exponentially with n
- The model was model-checked with different algorithms, execution times in seconds are given below (time limit = 12 hours)

n	BDD-based CTL MC	BDD-based LTL MC	Bounded MC (BMC)
7	39	485	95
8	26	197	191
9	24	166	339
10	17509	3525	604
11	731	> 43200	1035
12	71	3512	2176
13	1355	19897	3022
14	8130	> 43200	3784

When explicit-state model checking is better?

- Suppose that we have a closed-loop system where the plant is modeled as a state machine with a reasonably small number of states (e.g. ≤ 5000)
- Such plant models can be constructed automatically from traces (Buzhinsky et al. 2017)
- According to my practical experience, NuSMV processes large state machines poorly, making the benefits of the small state space impossible to exploit

- In contrast, explicit-state model checking is fast (compared to the open-loop case) when the state space of the plant model is small

Some practical results on model checking nuclear I&C systems (the NPP model was provided by Fortum)

		Subsystem	S1	S2	S3	S4	S5	S6	S7	S8
	# temporal specs		9	24	26	15	10	18	11	8
Open-loop time	Worse	SPIN	54	TL	TL	TL	TL	TL	3	8
	Better	NuSMV	5	1	11	11	1	21	1	2
Closed-loop time	Better	SPIN	3	44	277	98	256	148	3	3
	Worse	NuSMV	2611	137	769	TL	718	1104	268	8

- Model checking times are given in seconds
- Time limit (TL) $=$ 10 minutes $\times$ the number of temporal specifications
- Open-loop model checking is faster in NuSMV
- Closed-loop model checking is faster in SPIN

When both symbolic and explicit-state model checking fail

- If the model is too complex, NuSMV will be trying to verify the first temporal property for too long
- This can be so even with bounded model checking (BMC), a technique that checks temporal properties up to the given length of counterexamples

- In contrast, in SPIN the maximum search depth can be limited, leading to a possibility of performing a reduced, less reliable model checking
- Another technique with a similar effect is **bitstate hashing**, where the memory occupied by a single state is reduced