Lecture 5. Symbolic model checking. NuSMV and SPIN model checkers. User-friendly model checking

ELEC-E8110 Automation Systems Synthesis and Analysis

Igor Buzhinsky

igor.buzhinskii@aalto.fi

Aalto University
School of Electrical Engineering

2018

Symbolic model checking

Motivation

- State spaces can be very large

Motivation

- State spaces can be very large
- If there are ten 32-bit integer variables, how many states can the system have potentially?

Motivation

- State spaces can be very large
- If there are ten 32-bit integer variables, how many states can the system have potentially? $2^{320} \approx 2.1 \cdot 10^{96}$
- The so-called "state explosion" problem

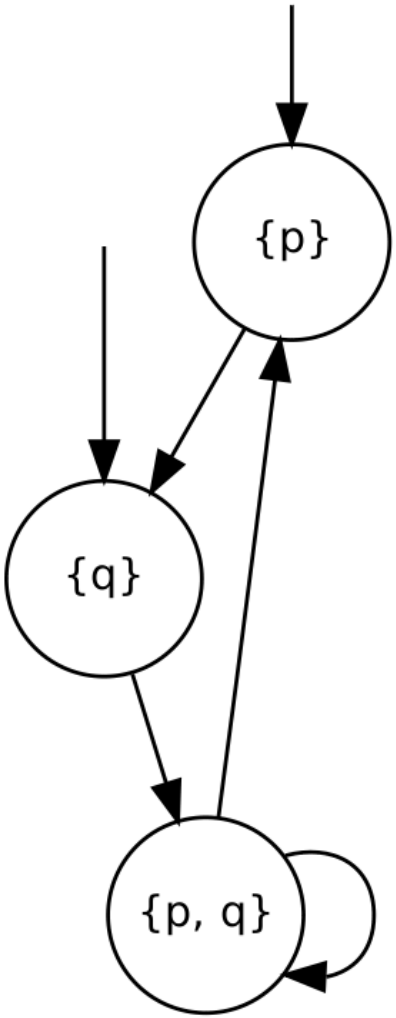
Motivation

- State spaces can be very large
- If there are ten 32-bit integer variables, how many states can the system have potentially? $2^{320} \approx 2.1 \cdot 10^{96}$
- The so-called "state explosion" problem
- PC can probably handle (store in memory and process) only about 10^9 states...

Motivation

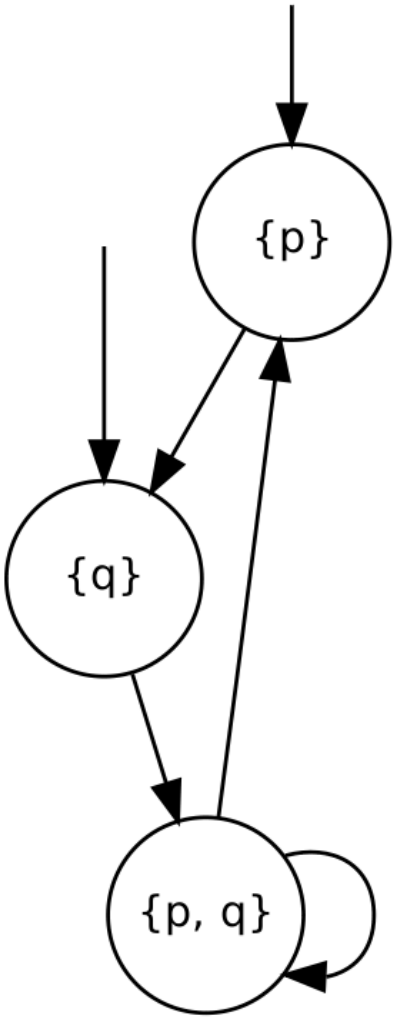
- State spaces can be very large
- If there are ten 32-bit integer variables, how many states can the system have potentially? $2^{320} \approx 2.1 \cdot 10^{96}$
- The so-called "state explosion" problem
- PC can probably handle (store in memory and process) only about 10^9 states...
- Can we avoid explicit construction of the state graph?

State subsets as Boolean constraints (1)



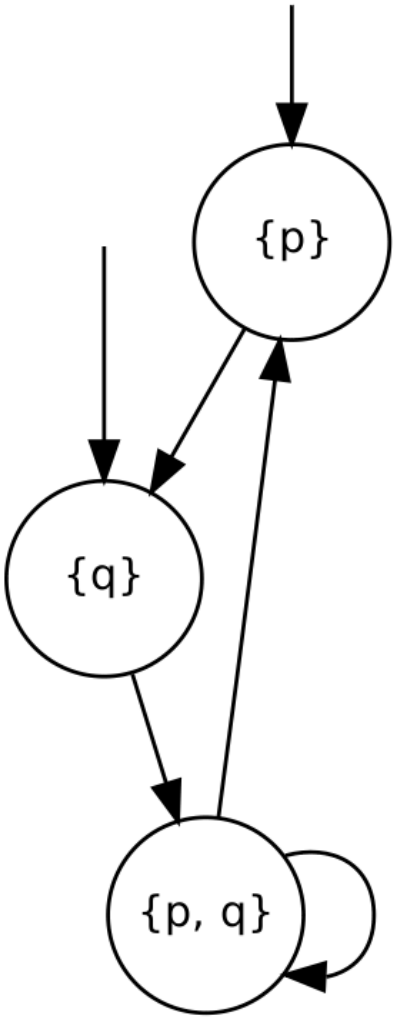
- Can you specify the set of reachable states as a Boolean formula?

State subsets as Boolean constraints (1)



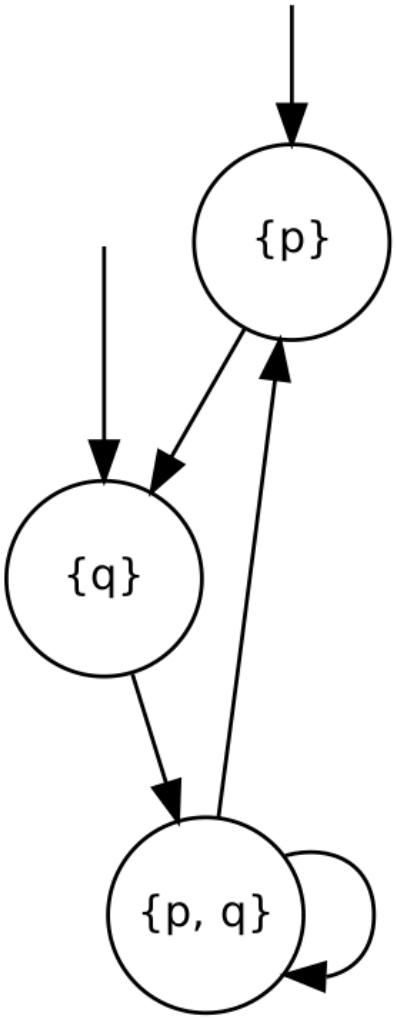
- Can you specify the set of reachable states as a Boolean formula?
- $p \vee q$

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?

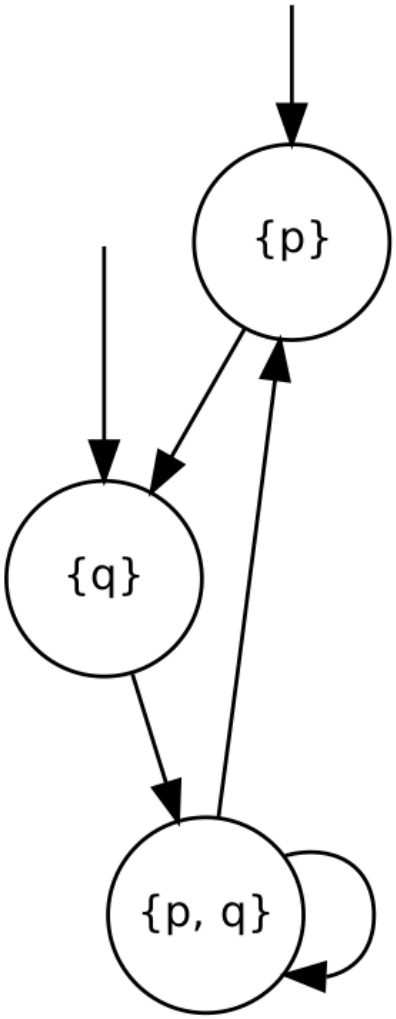
State subsets as Boolean constraints (1)



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

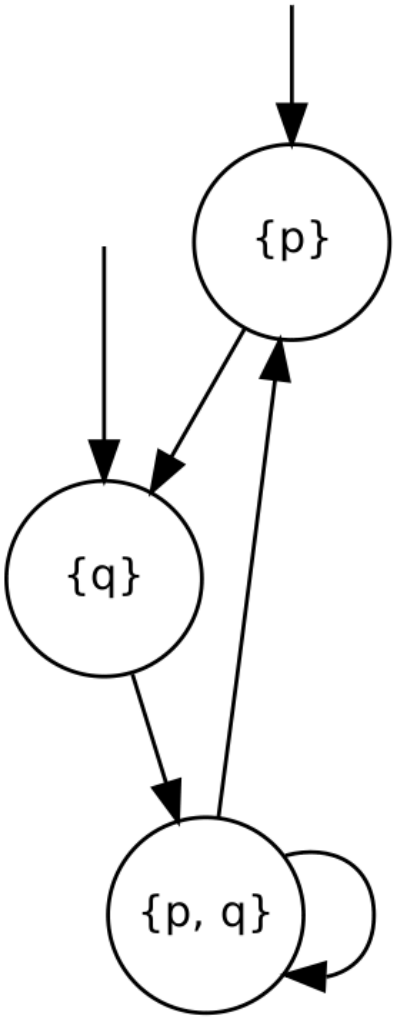
- What about only initial states?
- $p \oplus q = p \wedge \neg q \vee \neg p \wedge q$

State subsets as Boolean constraints (2)



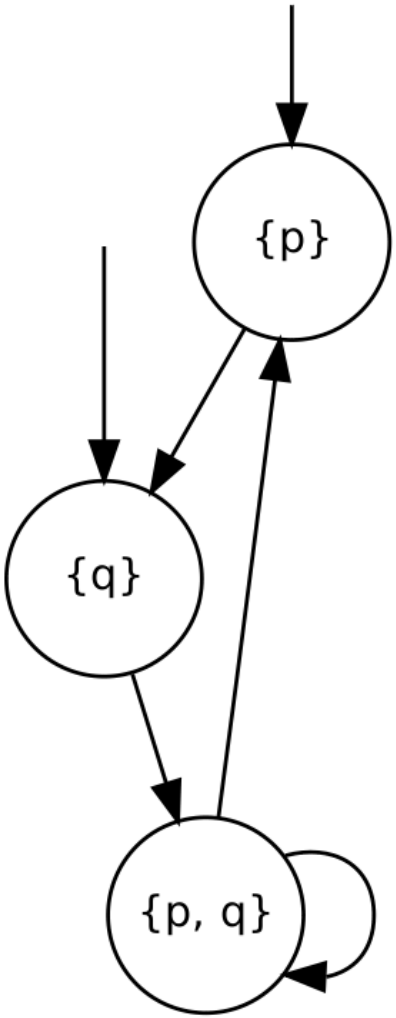
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step

State subsets as Boolean constraints (2)



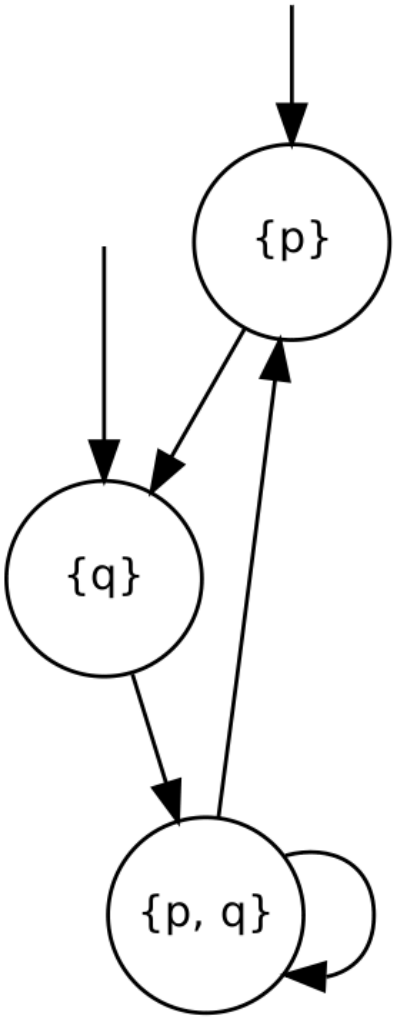
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula

State subsets as Boolean constraints (2)



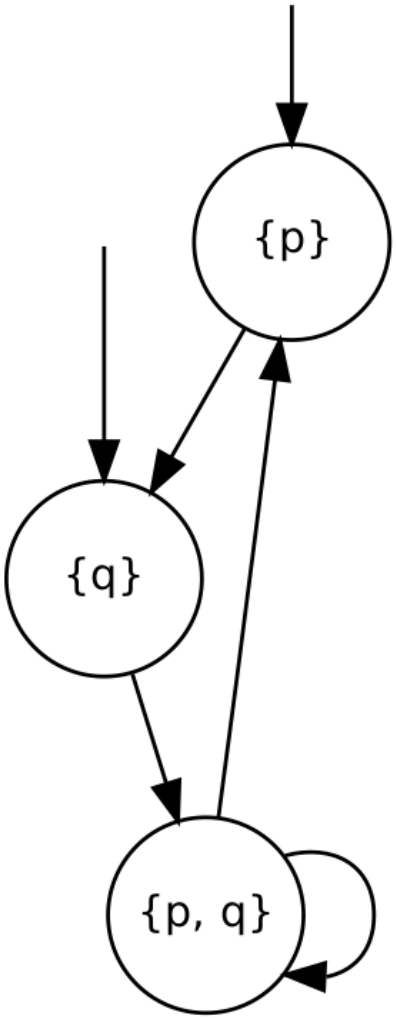
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
- $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge$

State subsets as Boolean constraints (2)



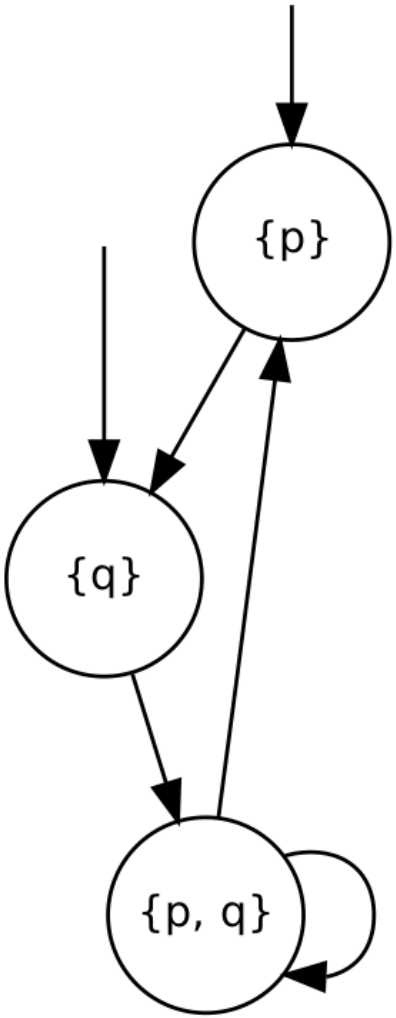
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
- $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge (q \wedge \neg p \rightarrow p' \wedge q')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

State subsets as Boolean constraints (2)



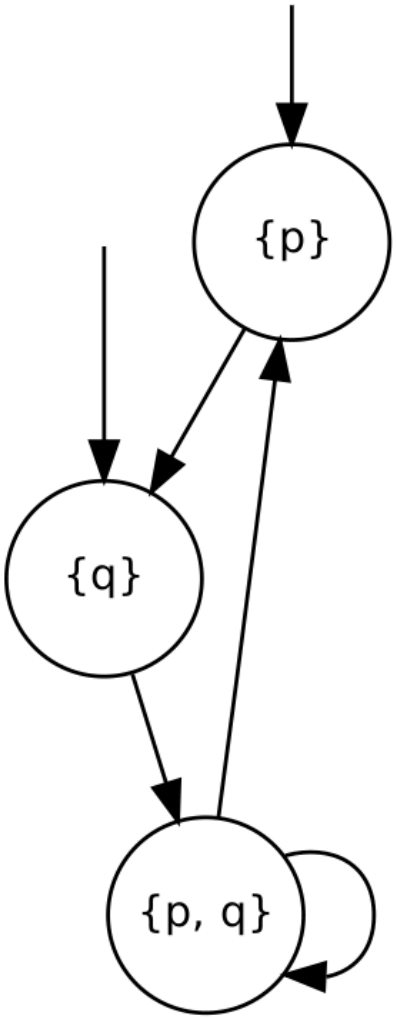
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
- $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge (q \wedge \neg p \rightarrow p' \wedge q')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

State subsets as Boolean constraints (2)



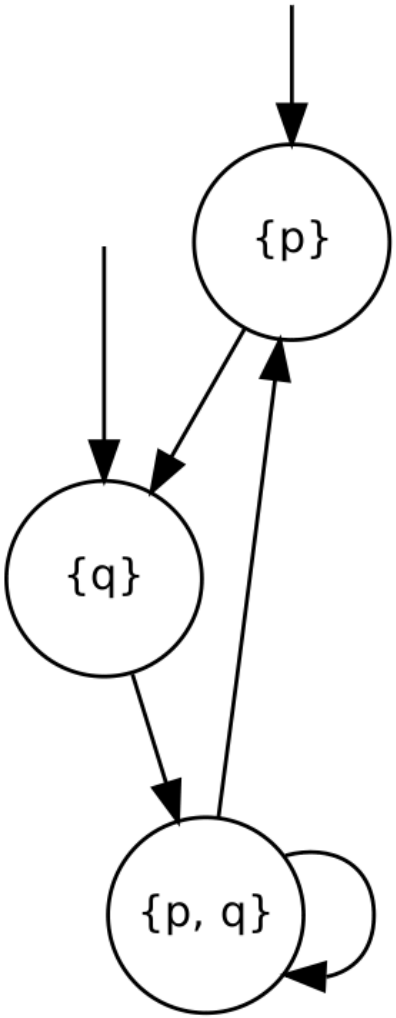
- What about the transition relation?
- p, q: values on this step
- p', q': values on the next step
- **Quiz:** specify the transition relation for the Kripke structure on the left as a Boolean formula
- $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge (q \wedge \neg p \rightarrow p' \wedge q')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

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
- $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge (q \wedge \neg p \rightarrow p' \wedge q') \wedge (p \wedge q \rightarrow p')$
- Alternative way: $(p \wedge \neg q \wedge q' \wedge \neg p') \vee$

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
- $(p \wedge \neg q \rightarrow q' \wedge \neg p') \wedge (q \wedge \neg p \rightarrow p' \wedge q') \wedge (p \wedge q \rightarrow p')$
- Alternative way: $(p \wedge \neg q \wedge q' \wedge \neg p') \vee (q \wedge \neg p \wedge p' \wedge q') \vee (p \wedge q \wedge p')$

Model checking with Boolean constraints?

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

Model checking with Boolean constraints?

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

Model checking with Boolean constraints?

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

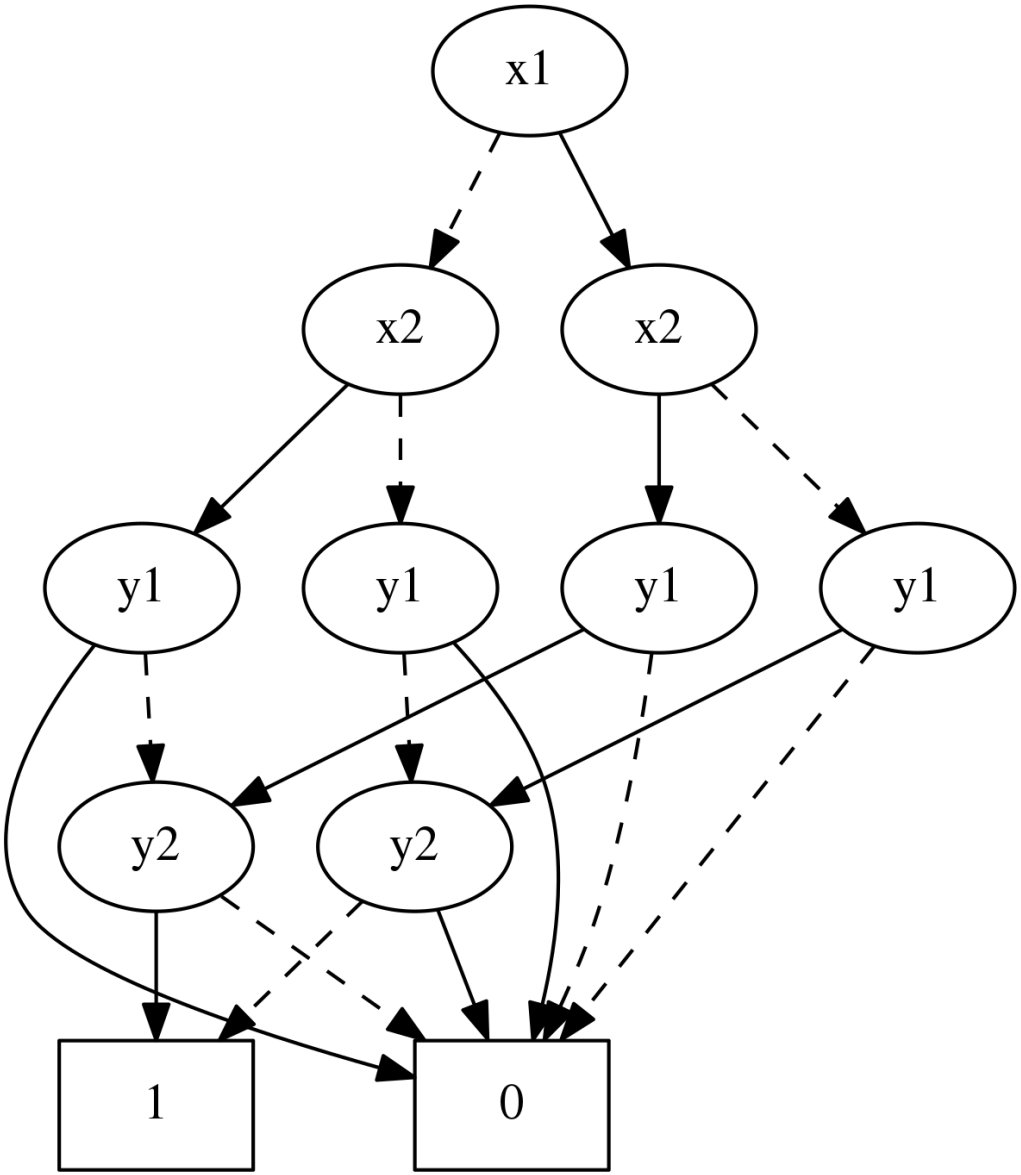
Model checking with Boolean constraints?

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

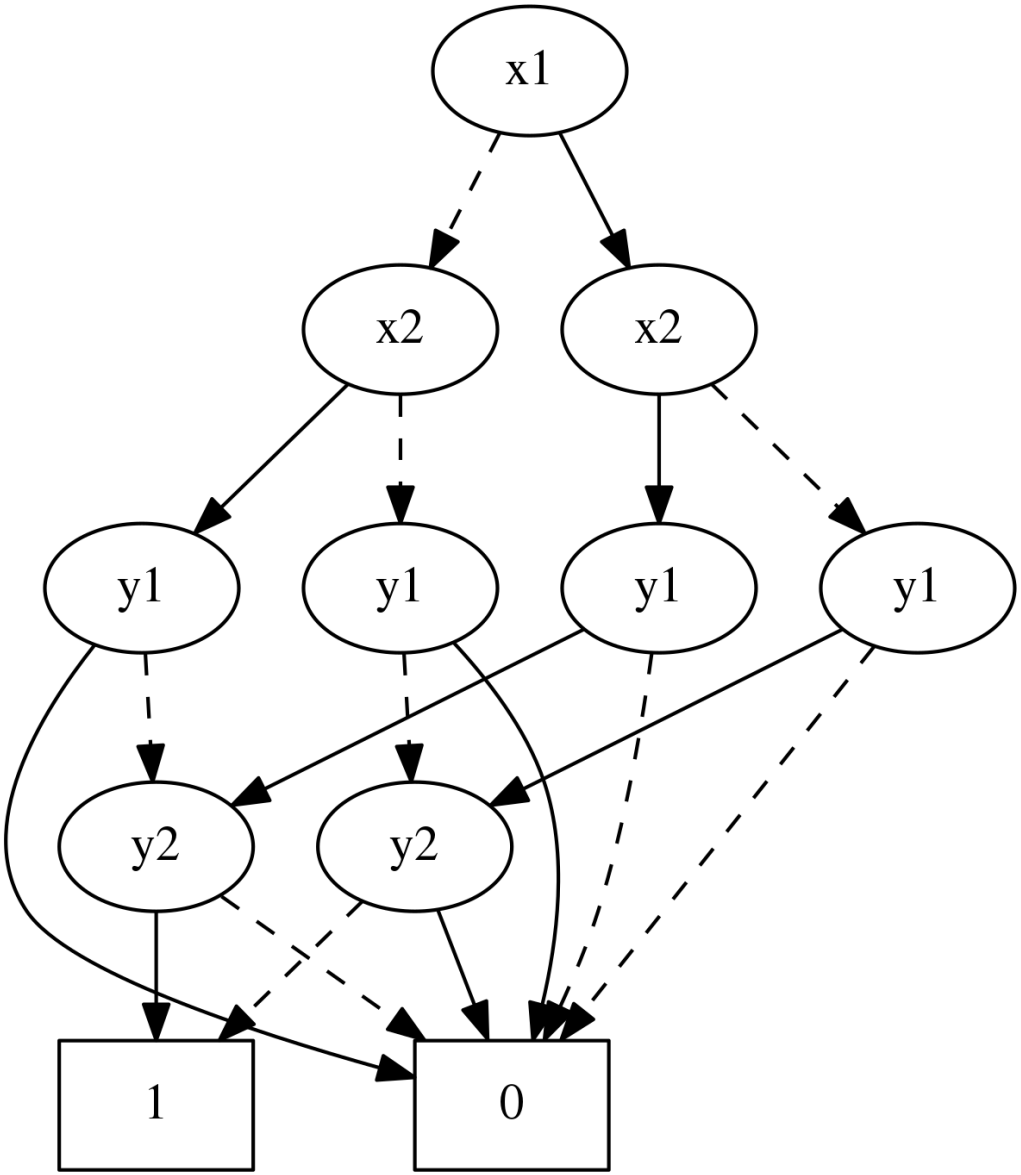
- Assume that our Kripke structure has atomic propositions $p_1, ..., p_n$
- Boolean constraints $f_{\text{init}}[p_1, ..., p_n]$ and $f_{\text{trans}}[p_1, ..., p_n, p'_1, ..., p'_n]$
- How to model-check $g = \mathbf{AG}\, h$, where h is a Boolean formula?
- Compute a sequence of formulae f_i: the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

NuSMV model checker

NuSMV

- Open-source symbolic model checker
- Supports LTL and CTL
- Can be downloaded here: `http://nusmv.fbk.eu/`
- Command-line tool, models are specified in text files
- If an LTL specification is false, the corresponding counterexample can be visualized with the tool `https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer`

NuSMV: example

```
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
INIT
    (c = 0)
  & (p)
TRANS
    (next(c) = (c + 1) mod 10)
  & (next(p) = !p)

CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))
```

- Integers are supported

NuSMV: example

```
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
INIT
    (c = 0)
  & (p)
TRANS
    (next(c) = (c + 1) mod 10)
  & (next(p) = !p)

CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))
```

- Integers are supported
- Are the specifications in the end satisfied?

NuSMV: example

```
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
INIT
    (c = 0)
  & (p)
TRANS
    (next(c) = (c + 1) mod 10)
  & (next(p) = !p)

CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))
```

- Integers are supported
- Are the specifications in the end satisfied? – Yes

NuSMV: example

```
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
INIT
    (c = 0)
  & (p)
TRANS
    (next(c) = (c + 1) mod 10)
  & (next(p) = !p)

CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))
```

- Integers are supported
- Are the specifications in the end satisfied? – Yes
- What about q?

NuSMV: example

```
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
INIT
    (c = 0)
  & (p)
TRANS
    (next(c) = (c + 1) mod 10)
  & (next(p) = !p)

CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))
```

- Integers are supported
- Are the specifications in the end satisfied? – Yes
- What about q? – It can be anything on any turn

NuSMV: alternative syntax

```
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
ASSIGN
    init(c) := 0;
    init(p) := TRUE;
    next(c) := c_plus_1 mod 10;
    next(p) := !p;
DEFINE
    c_plus_1 := c + 1;

CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))
```

- Explicit definitions for values changes
- Sub-expressions can be defined and reused
- Assignments can be nondeterministic, e.g. `init(c) := {0, 1};`
- INIT, TRANS, ASSIGN and DEFINE can co-exist

NuSMV: alternative syntax

```
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
ASSIGN
    init(c) := 0;
    init(p) := TRUE;
    next(c) := c_plus_1 mod 10;
    next(p) := !p;
DEFINE
    c_plus_1 := c + 1;

CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))
```

- Explicit definitions for values changes
- Sub-expressions can be defined and reused
- Assignments can be nondeterministic, e.g. `init(c) := {0, 1};`
- INIT, TRANS, ASSIGN and DEFINE can co-exist
- Use ASSIGN and DEFINE instead of INIT and TRANS where possible! It is easy to make a modeling error with INIT and TRANS

NuSMV: cylinder

```
MODULE CYLINDER(fwd, back)
VAR
    pos: 0..5;
ASSIGN
    init(pos) := 0;
    next(pos) := fwd ? next_pos : back ? prev_pos : pos;
DEFINE
    next_pos := pos < 5 ? (pos + 1) : pos;
    prev_pos := pos > 0 ? (pos - 1) : pos;
    home := pos = 0;
    end := pos = 5;
```

- Modules can have inputs (in the declaration), and their variables and definitions can be interpreted as outputs
- C-style choice operator ?:

NuSMV: controller

```
MODULE CONTROLLER(home, end)
VAR
    state: {moving_fwd, moving_back};
ASSIGN
    init(state) := moving_fwd;
    next(state) := case
        home: moving_fwd;
        end: moving_back;
        TRUE: state;
    esac;
DEFINE
    fwd := state = moving_fwd;
    back := state = moving_back;
```

- Example of explicit state machine modeling

NuSMV: closed-loop composition

```
MODULE main()
VAR
    -- this is the way to write comments, by the way
    cyl: CYLINDER(ctr.fwd, ctr.back);
    ctr: CONTROLLER(cyl.home, cyl.end);

LTLSPEC G F cyl.end -- TRUE
LTLSPEC G F cyl.home -- TRUE
```

- **Synchronous**: all the modules make a step together!

NuSMV: closed-loop composition

```
MODULE main()
VAR
    -- this is the way to write comments, by the way
    cyl: CYLINDER(ctr.fwd, ctr.back);
    ctr: CONTROLLER(cyl.home, cyl.end);

LTLSPEC G F cyl.end -- TRUE
LTLSPEC G F cyl.home -- TRUE
```

- **Synchronous**: all the modules make a step together!
- How to model asynchronous interaction?

SPIN model checker

SPIN

- Open-source **explicit-state** model checker
- Supports LTL
- Can be downloaded here: `http://spinroot.com/`
- Can be run as a command-line tool, but also has GUI (iSpin)

- Will not be covered in tutorials, assignments and the exam
- You can try it yourself if you are interested

SPIN: cylinder

```
int pos = 0;
bool home = true, end, fwd, back;

// to be executed in a loop:
#define next_pos (pos < 5 -> (pos + 1) : pos)
#define prev_pos (pos > 0 -> (pos - 1) : pos)
pos = (fwd -> next_pos : (back -> prev_pos : pos));
home = pos == 0;
end = pos == 5;
```

- C-like syntax, but the choice operator has a different syntax
- C macros and other preprocessor directives are supported
- Conditional and loop statements (not shown) are **very** different, see online manuals if interested

SPIN: controller

```
bool home = true, end, fwd, back;
mtype = { moving_fwd, moving_back };
mtype state;

// to be executed in a loop:
state = (home -> moving_fwd :
        (end -> moving_back : state));
fwd = state == moving_fwd;
back = state == moving_back;
```

- mtype can be used for enumerations

SPIN: closed-loop composition

```
int pos = 0;
bool home = true, end, fwd, back;
mtype = { moving_fwd, moving_back };
mtype state;

init { do :: atomic { // a loop of atomic steps
        // <plant loop code>
        // <controller loop code>
} od }

ltl visiting_end  { [] <> end }; // G F end, true
ltl visiting_home { [] <> home }; // G F home, true
```

- Using this pattern, PLC-like applications can be modeled

Like UPPAAL, SPIN can verify asynchronous applications

- Multiple processes are supported
- `init` is executed in the beginning
- Other process types can be declared with the keyword `proctype`
- Their instances can be spawned with the keyword `run`
- Processes can execute asynchronously, unless explicitly constrained (e.g. by channels)
- *Partial order reduction* is used to reduce the state space in case of asynchrony

Literature

1. Baier, C., & Katoen, J. P. (2008). Principles of model checking. MIT press.
2. Clarke, E. M., Grumberg, O., & Peled, D. (1999). Model checking. MIT press.
3. 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), 142–170.
4. NuSMV tutorial: `http://nusmv.fbk.eu/NuSMV/tutorial/`
5. SPIN online references: `http://spinroot.com/spin/Man/`

User-friendly model checking

Why is it difficult to adopt model checking in industry?

Why is it difficult to adopt model checking in industry?

- **Efforts** of formal modeling

Why is it difficult to adopt model checking in industry?

- **Efforts** of formal modeling
- **Human factor** during modeling

Why is it difficult to adopt model checking in industry?

- **Efforts** of formal modeling
- **Human factor** during modeling
- **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 adopt model checking in industry?

- **Efforts** of formal modeling
- **Human factor** during modeling
- **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 still be problematic for symbolic model checkers

Why is it difficult to adopt model checking in industry?

- **Efforts** of formal modeling
- **Human factor** during modeling
- **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 still be problematic for symbolic model checkers

- **Knowledge and experience are required to use formal methods correctly and efficiently**

Why is it difficult to adopt model checking in industry?

- **Efforts** of formal modeling
- **Human factor** during modeling
- **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 still be problematic for symbolic model checkers

- **Knowledge and experience are required to use formal methods correctly and efficiently**
- How to mitigate this problem?

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

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

LTL P is false:

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

- "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

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.

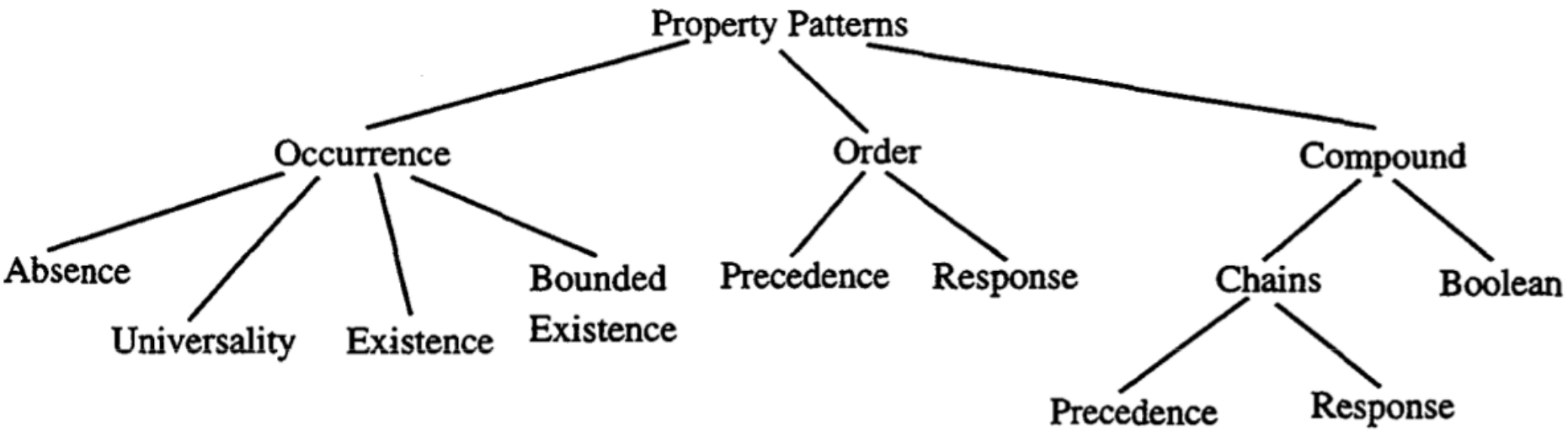
CTL S responds to P:

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

LTL S responds to P:

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

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

Literature

1. 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
2. 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
3. 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