Project 2: 53 days left

Prevention-Based Cyber Defense: Formal Methods

CS 459/559: Science of Cyber Security
13th Lecture

Instructor:

Guanhua Yan

Agenda

- ~~Quiz 1: September 29 (closed book)~~
- ~~Project 1 (offense): October 10~~
- Project 2 (defense): December 5
- Presentations: 11/17, 11/19, 11/24, 12/1, 12/3
- Final report: December 15

Outline

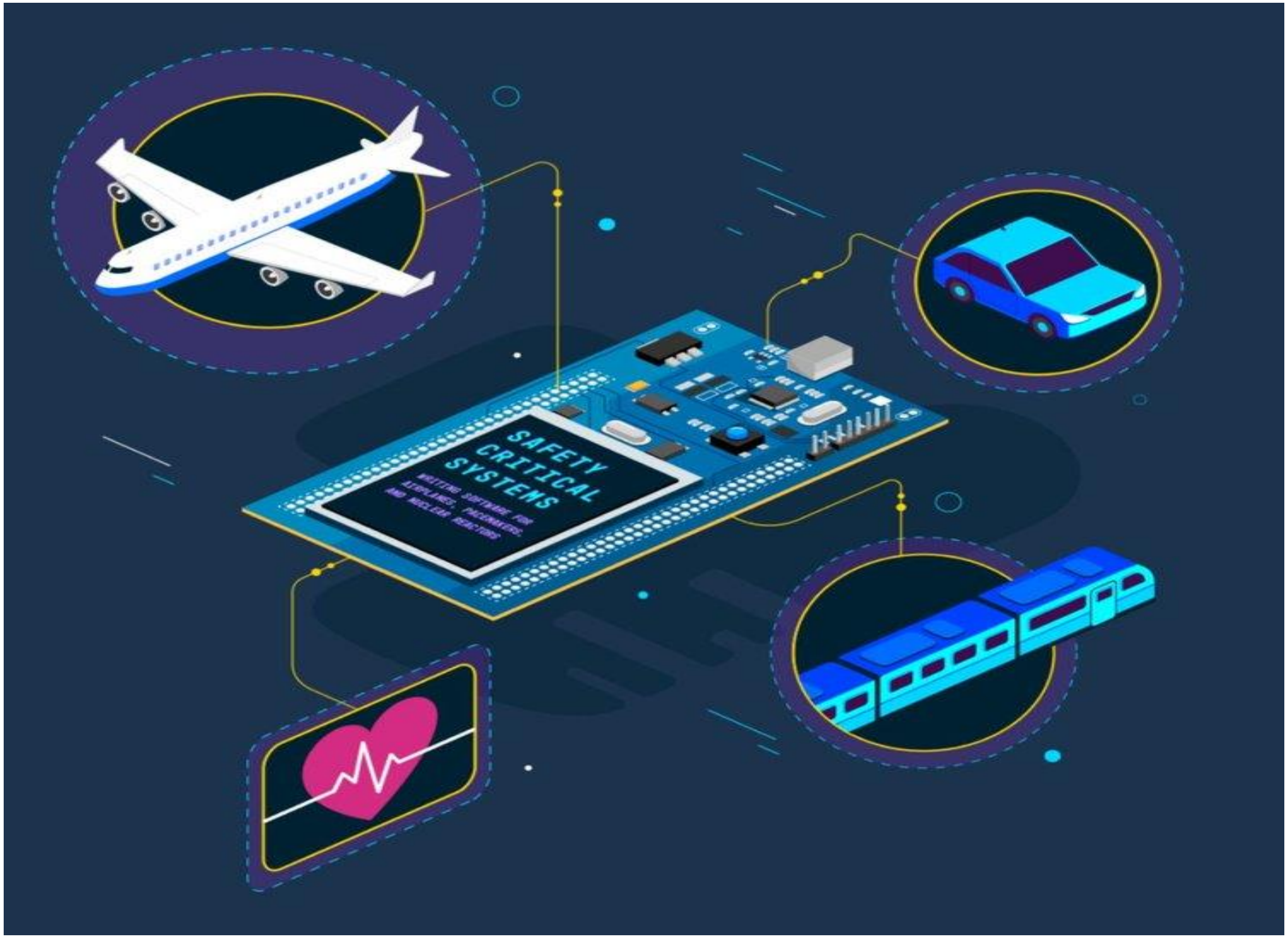
- **Formal verification**

- **Model checking**
  - Introduction
  - Models
  - Specifications
  - Algorithms

- *The NuSMV Model Checker*

Formal Verification

Safety-critical systems



Need for Formal Methods

- System reliability increasingly depends on the “correct” functioning of hardware and software
- Difficulties with traditional methodologies:
  - Ambiguities (in requirements, architecture, detailed design).
  - Errors in specification/design refinements.
  - The assurance provided by testing can be too limited.
  - Testing came too late in development
- Consequences:
  - Expensive errors in the early design phases.
  - Infeasibility of achieving high reliability requirements.
  - Low software quality: poor documentation, limited modifiability

Need for Formal Methods

- System reliability increasingly depends on the “correct” functioning of hardware and software
- Difficulties with traditional methodologies:
  - Ambiguities (in requirements, architecture, detailed design).
  - Errors in specification/design refinements.
  - The assurance provided by testing can be too limited.
  - Testing came too late in development
- Consequences:
  - Expensive errors in the early design phases.
  - Infeasibility of achieving high reliability requirements.
  - Low software quality: poor documentation, limited modifiability

Need for Formal Methods

- System reliability increasingly depends on the “correct” functioning of hardware and software
- Difficulties with traditional methodologies:
  - Ambiguities (in requirements, architecture, detailed design).
  - Errors in specification/design refinements.
  - The assurance provided by testing can be too limited.
  - Testing came too late in development
- Consequences:
  - Expensive errors in the early design phases.
  - Infeasibility of achieving high reliability requirements.
  - Low software quality: poor documentation, limited modifiability

Formal Methods: Solution and Benefits

- **Formal Methods:**
  - Formal Specification: precise, unambiguous description.
  - Formal Validation & Verification Tools: exhaustive analysis of the formal specification.

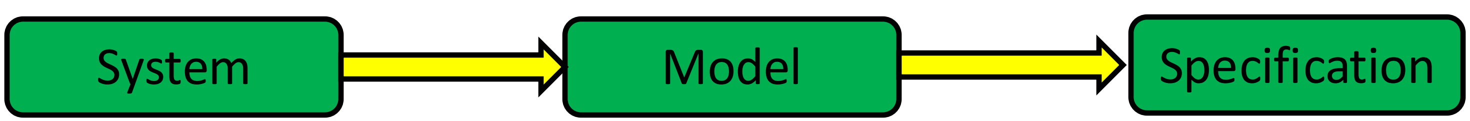
- **Potential Benefits:**
  - Find design bugs in early design stages.
  - Achieve higher quality standards.
  - Shorten time-to-market, reducing manual validation phases.
  - Produce well documented, maintainable products

Verification vs. validation

- **Verification**:
  - "Are we building the product right?"
  - Software verification: the software should conform to its specification.
- **Validation**:
  - "Are we building the right product?"
  - Software validation: the software should do what the user really needs / wants.

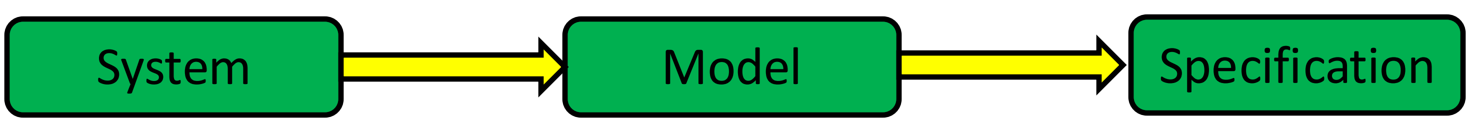
Formal Verification (in a nutshell)

- Create a *formal model* of some system of interest
  - Hardware
  - Communication protocol
  - Software, esp. concurrent software
- Describe formally a *specification* that we desire the model to satisfy
- Check the model satisfies the specification
  - theorem proving (usually interactive but not necessarily)
  - Model checking



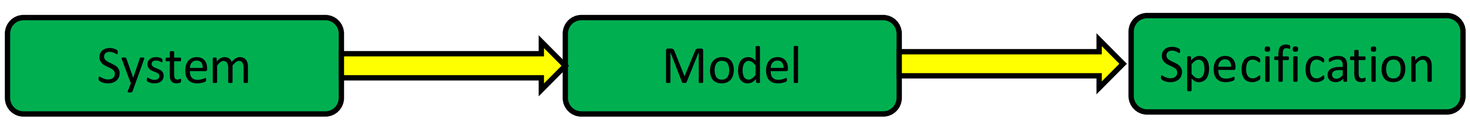
Formal Verification (in a nutshell)

- Create a *formal model* of some system of interest
  - Hardware
  - Communication protocol
  - Software, esp. concurrent software
- Describe formally a *specification* that we desire the model to satisfy
- Check the model satisfies the specification
  - theorem proving (usually interactive but not necessarily)
  - Model checking



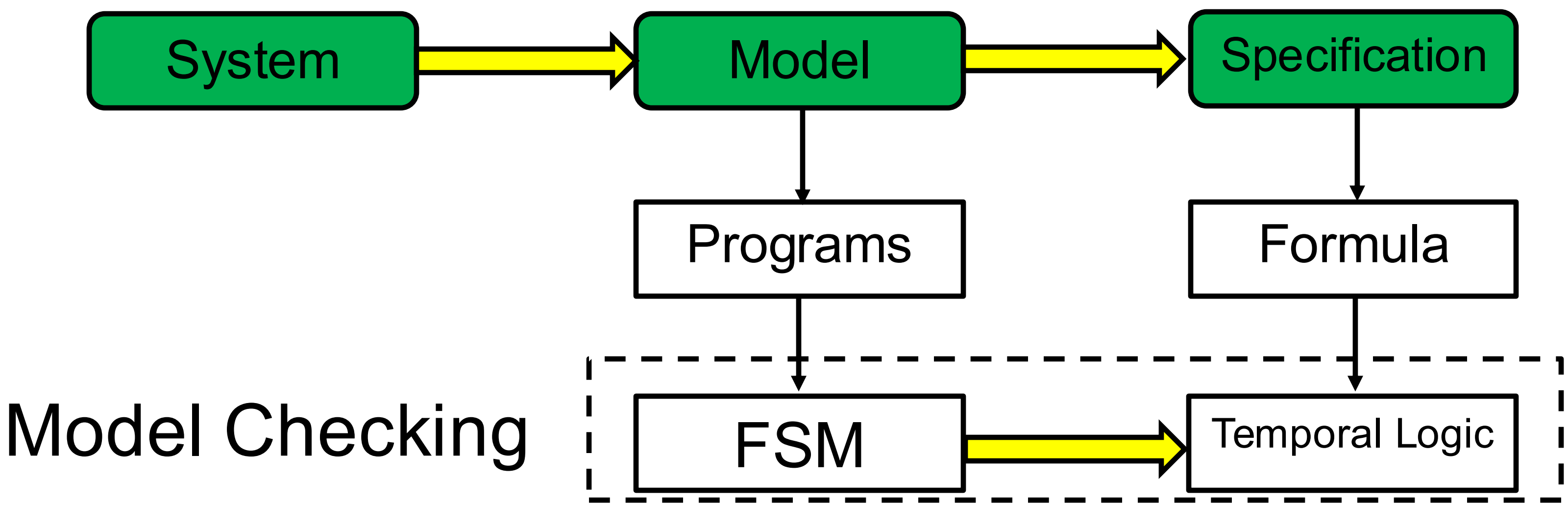
Formal Verification (in a nutshell)

- Create a *formal model* of some system of interest
  - Hardware
  - Communication protocol
  - Software, esp. concurrent software
- Describe formally a *specification* that we desire the model to satisfy
- Check the model satisfies the specification
  - theorem proving (usually interactive but not necessarily)
  - Model checking



Introduction to Model Checking

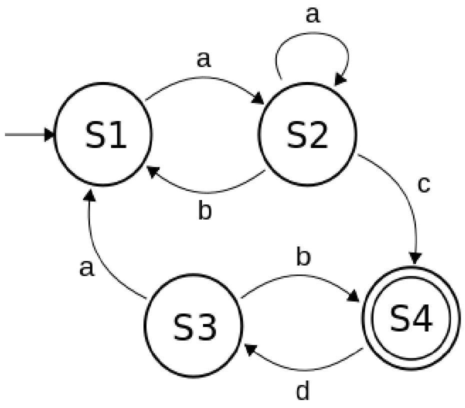
- Specifications as Formulas, Programs as Models
- Programs are abstracted as Finite State Machines
- Formulas are in Temporal Logic



Model checking: models

FSM: Finite State Machine

A Finite State Machine (FSM) is defined as a 5-tuple (Q, Σ, δ, q_0, F) where,



(1) Q is a finite set of *states*.

(2) Σ is a finite set of symbols or the *alphabet*.

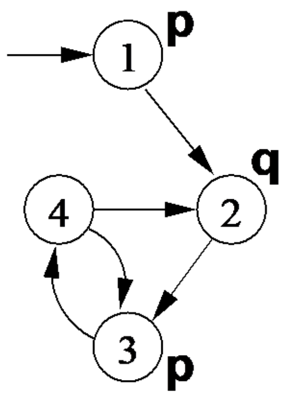
(3) δ: Q x Σ -> Q is the *transition function*

(4) q_0 is an element of Q called the *start state*, and

(5) F is a subset of Q called the set of accept *states*.

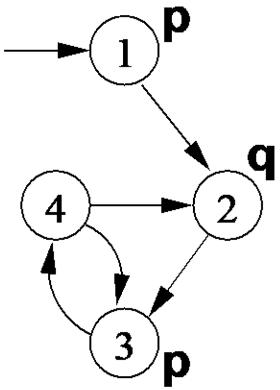
Modeling the system: Kripke models

- Kripke models are used to describe reactive systems:
  - nonterminating systems with infinite behaviors,
  - e.g. communication protocols, operating systems, hardware circuits;
  - represent dynamic evolution of modeled systems;
  - values to state variables, program counters, content of communication channels.
- Formally, a Kripke model (S, R, I, L) consists of
  - a set of states S;
  - a set of initial states $I \subseteq S$;
  - a set of transitions $R \subseteq S \times S$;
  - a labeling $L \subseteq S \times AP$.



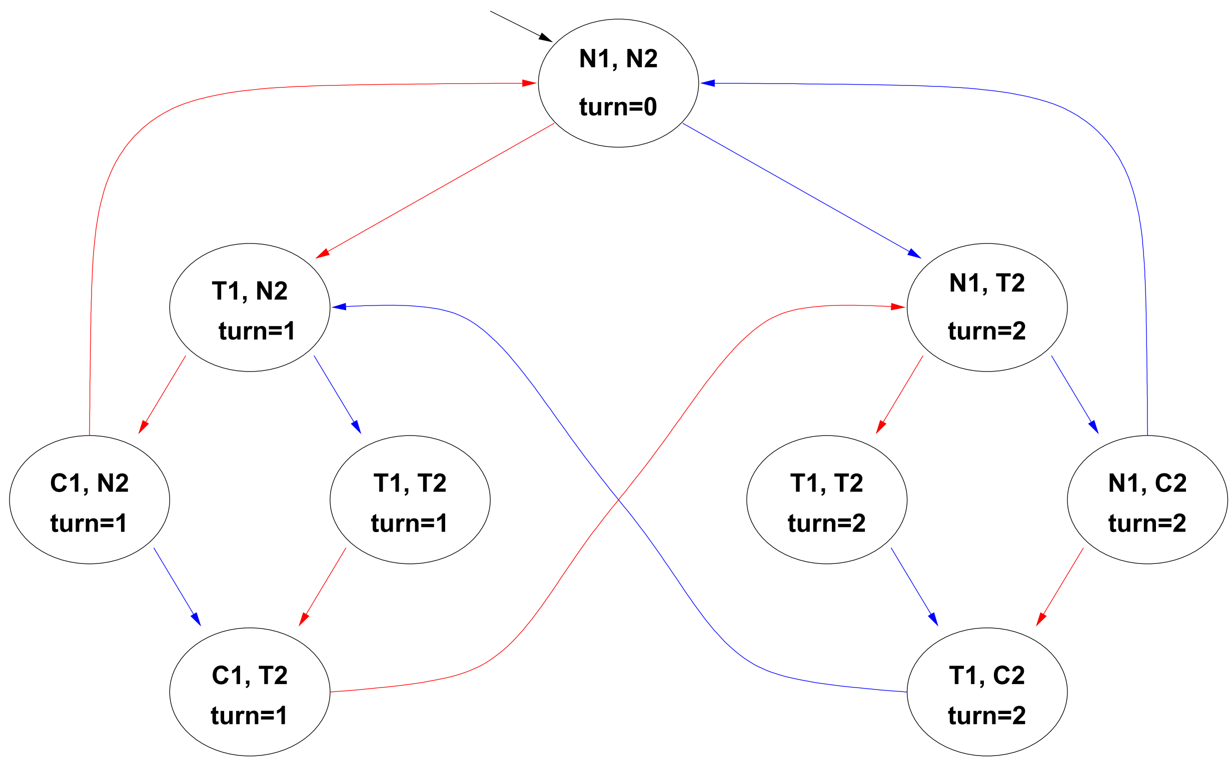
Modeling the system: Kripke models

- Kripke models are used to describe reactive systems:
  - nonterminating systems with infinite behaviors,
  - e.g. communication protocols, operating systems, hardware circuits;
  - represent dynamic evolution of modeled systems;
  - values to state variables, program counters, content of communication channels.
- Formally, a Kripke model (S, R, I, L) consists of
  - a set of states S;
  - a set of initial states $I \subseteq S$;
  - a set of transitions $R \subseteq S \times S$;
  - a labeling $L \subseteq S \times AP$. ★



AP: set of atomic propositions

A Kripke model for mutual exclusion



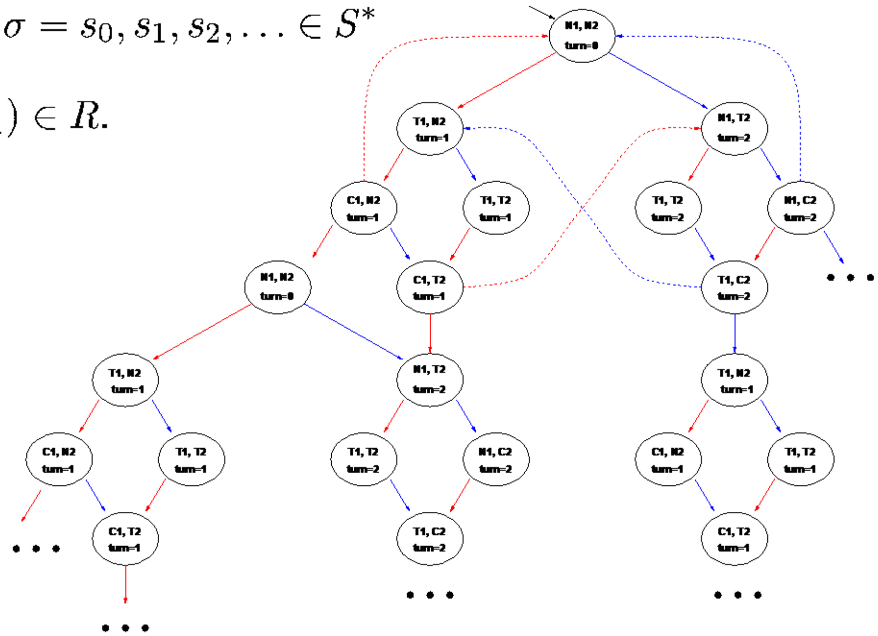
N = noncritical, T = trying, C = critical User 1 User 2

Path in a Kripke Model

- A path in a Kripke model M is an infinite sequence

$$\sigma = s_0, s_1, s_2, \ldots \in S^*$$

such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$.



- A state s is reachable in M if there is a path from the initial states to s.

Model checking: specifications

Model Checking – Specifications

We are interested in specifying behaviours of systems over time.

- Use **Temporal Logic**

Specifications are built from:

1. Primitive properties of individual states
   e.g., "is on", "is off", "is active", "is reading";
2. propositional connectives $\wedge, \vee, \neg, \rightarrow$;
3. and **temporal** connectives: *e.g.*,
   At **all times**, the system is not simultaneously *reading* and *writing*.
   If a *request* signal is asserted **at some time**, a corresponding *grant* signal will be asserted **within 10 time units**.

The exact set of temporal connectives differs across temporal logics.
Logics can differ in how they treat time:

- **Linear time** vs. **Branching time**

These differ in reasoning about *non-determinism*.

Non-determinism

In general, system descriptions are *non-deterministic.*

A system is *non-deterministic* when, from some state there are **multiple** alternative next states to which the system could transition.

Non-determinism is good for:

- Modelling alternative inputs to the system from its environment (*External non-determinism*)
- Under-specifying the model, allowing it to capture many possible system implementations (*Internal non-determinism*)

Linear vs. Branching Time

- **Linear Time**
  - Considers paths (sequences of states)
  - If system is non-deterministic, many paths for each initial state
  - Questions of the form:
    - For all paths, does some path property hold?
    - Does there exist a path such that some path property holds?
- **Branching Time**
  - Considers tree of possible future states from each initial state
  - If system is non-deterministic from some state, tree forks
  - Questions can become more complex, *e.g.*,
    - For all states reachable from an initial state, does there exist an onwards path to a state satisfying some property?

Linear Time Temporal Logic (LTL)

LTL properties are evaluated over paths, i.e., over infinite, linear sequences of states:

$$s[0] \rightarrow s[1] \rightarrow \cdots \rightarrow s[t] \rightarrow s[t+1] \rightarrow \cdots$$

LTL provides the following temporal operators:

- "Finally" (or "future"): Fp is true in $s[t]$ iff p is true in **some** $s[t']$ with $t' \geq t$
- "Globally" (or "always"): Gp is true in $s[t]$ iff p is true in **all** $s[t']$ with $t' \geq t$
- "Next": Xp is true in $s[t]$ iff p is true in $s[t+1]$
- "Until": pUq is true in $s[t]$ iff
  - q is true in some state $s[t']$ with $t' \geq t$
  - p is true in all states $s[t'']$ with $t \leq t'' < t'$

Linear Time Temporal Logic (LTL)

LTL properties are evaluated over paths, i.e., over infinite, linear sequences of states:

$$s[0] \rightarrow s[1] \rightarrow \cdots \rightarrow s[t] \rightarrow s[t+1] \rightarrow \cdots$$

LTL provides the following temporal operators:

- "Finally" (or "future"): Fp is true in $s[t]$ iff p is true in **some** $s[t']$ with $t' \geq t$
- "Globally" (or "always"): Gp is true in $s[t]$ iff p is true in **all** $s[t']$ with $t' \geq t$
- "Next": Xp is true in $s[t]$ iff p is true in $s[t+1]$
- "Until": pUq is true in $s[t]$ iff
  - q is true in some state $s[t']$ with $t' \geq t$
  - p is true in all states $s[t'']$ with $t \leq t'' < t'$

Linear Time Temporal Logic (LTL)

LTL properties are evaluated over paths, i.e., over infinite, linear sequences of states:

$$s[0] \to s[1] \to \cdots \to s[t] \to s[t+1] \to \cdots$$

LTL provides the following temporal operators:

- "Finally" (or "future"): Fp is true in $s[t]$ iff p is true in **some** $s[t']$ with $t' \geq t$
- "Globally" (or "always"): Gp is true in $s[t]$ iff p is true in **all** $s[t']$ with $t' \geq t$
- "Next": Xp is true in $s[t]$ iff p is true in $s[t+1]$
- "Until": pUq is true in $s[t]$ iff
  - q is true in some state $s[t']$ with $t' \geq t$
  - p is true in all states $s[t'']$ with $t \leq t'' < t'$

Linear Time Temporal Logic (LTL)

LTL properties are evaluated over paths, i.e., over infinite, linear sequences of states:

$$s[0] \rightarrow s[1] \rightarrow \cdots \rightarrow s[t] \rightarrow s[t+1] \rightarrow \cdots$$

LTL provides the following temporal operators:

- "Finally" (or "future"): Fp is true in $s[t]$ iff p is true in **some** $s[t']$ with $t' \geq t$
- "Globally" (or "always"): Gp is true in $s[t]$ iff p is true in **all** $s[t']$ with $t' \geq t$
- "Next": Xp is true in $s[t]$ iff p is true in $s[t+1]$
- "Until": pUq is true in $s[t]$ iff
  - q is true in some state $s[t']$ with $t' \geq t$
  - p is true in all states $s[t'']$ with $t \leq t'' < t'$

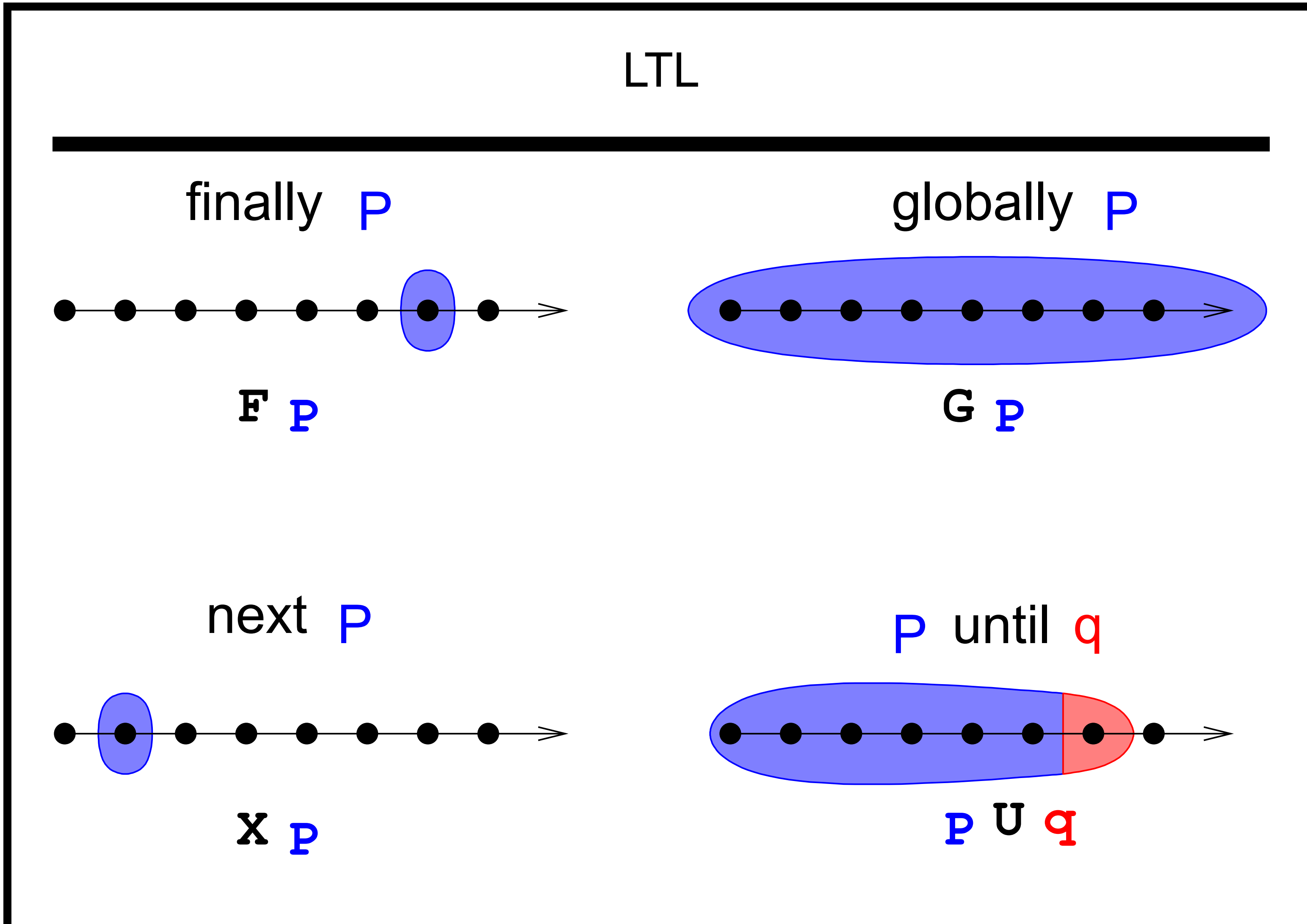
Linear Time Temporal Logic (LTL)

LTL properties are evaluated over paths, i.e., over infinite, linear sequences of states:

$$s[0] \rightarrow s[1] \rightarrow \cdots \rightarrow s[t] \rightarrow s[t+1] \rightarrow \cdots$$

LTL provides the following temporal operators:

- "Finally" (or "future"): Fp is true in $s[t]$ iff p is true in **some** $s[t']$ with $t' \geq t$
- "Globally" (or "always"): Gp is true in $s[t]$ iff p is true in **all** $s[t']$ with $t' \geq t$
- "Next": Xp is true in $s[t]$ iff p is true in $s[t+1]$
- "Until": pUq is true in $s[t]$ iff
  - q is true in some state $s[t']$ with $t' \geq t$
  - p is true in all states $s[t'']$ with $t \leq t'' < t'$


LTL
finally p
F p
globally p
G p
next p
X p
p until q
p U q

A Taste of LTL – Examples

1. $\mathbf{G}\ invariant$

   invariant is true for all future positions
2. $\mathbf{G}\ \neg(read \wedge write)$

   In all future positions, it is not the case that *read* and *write*
3. $\mathbf{G}(request \rightarrow \mathbf{F} grant)$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.
4. $\mathbf{G}(request \rightarrow (request\ \mathbf{U}\ grant))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.
5. $\mathbf{G}\ \mathbf{F}\ enabled$

   In all future positions, there is a future position where *enabled* holds.
6. $\mathbf{F}\ \mathbf{G}\ enabled$

   There is a future position, from which all future positions have *enabled* holding.

A Taste of LTL – Examples

1. $\mathbf{G}\ \mathit{invariant}$

   invariant is true for all future positions

2. $\mathbf{G}\ \neg(\mathit{read} \wedge \mathit{write})$

   In all future positions, it is not the case that *read* and *write*

3. $\mathbf{G}(\mathit{request} \rightarrow \mathbf{F}\mathit{grant})$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.

4. $\mathbf{G}(\mathit{request} \rightarrow (\mathit{request}\ \mathbf{U}\ \mathit{grant}))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.

5. $\mathbf{G}\ \mathbf{F}\ \mathit{enabled}$

   In all future positions, there is a future position where *enabled* holds.

6. $\mathbf{F}\ \mathbf{G}\ \mathit{enabled}$

   There is a future position, from which all future positions have *enabled* holding.

A Taste of LTL – Examples

1. $\mathbf{G}\ \mathit{invariant}$

   invariant is true for all future positions
2. $\mathbf{G}\ \neg(\mathit{read} \wedge \mathit{write})$

   In all future positions, it is not the case that *read* and *write*
3. $\mathbf{G}(\mathit{request} \rightarrow \mathbf{F}\mathit{grant})$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.
4. $\mathbf{G}(\mathit{request} \rightarrow (\mathit{request}\ \mathbf{U}\ \mathit{grant}))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.
5. $\mathbf{G}\ \mathbf{F}\ \mathit{enabled}$

   In all future positions, there is a future position where *enabled* holds.
6. $\mathbf{F}\ \mathbf{G}\ \mathit{enabled}$

   There is a future position, from which all future positions have *enabled* holding.

A Taste of LTL – Examples

1. $\mathbf{G}\ invariant$

   invariant is true for all future positions
2. $\mathbf{G}\ \neg(read \wedge write)$

   In all future positions, it is not the case that *read* and *write*
3. $\mathbf{G}(request \rightarrow \mathbf{F} grant)$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.
4. $\mathbf{G}(request \rightarrow (request\ \mathbf{U}\ grant))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.
5. $\mathbf{G}\ \mathbf{F}\ enabled$

   In all future positions, there is a future position where *enabled* holds.
6. $\mathbf{F}\ \mathbf{G}\ enabled$

   There is a future position, from which all future positions have *enabled* holding.

A Taste of LTL – Examples

1. $\mathbf{G}\ invariant$

   invariant is true for all future positions

2. $\mathbf{G}\ \neg(read \wedge write)$

   In all future positions, it is not the case that *read* and *write*

3. $\mathbf{G}(request \rightarrow \mathbf{F} grant)$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.

4. $\mathbf{G}(request \rightarrow (request\ \mathbf{U}\ grant))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.

5. $\mathbf{G}\ \mathbf{F}\ enabled$

   In all future positions, there is a future position where *enabled* holds.

6. $\mathbf{F}\ \mathbf{G}\ enabled$

   There is a future position, from which all future positions have *enabled* holding.

A Taste of LTL – Examples

1. $\mathbf{G}\ invariant$

   invariant is true for all future positions
2. $\mathbf{G}\ \neg(read \wedge write)$

   In all future positions, it is not the case that *read* and *write*
3. $\mathbf{G}(request \rightarrow \mathbf{F} grant)$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.
4. $\mathbf{G}(request \rightarrow (request\, \mathbf{U}\ grant))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.
5. $\mathbf{G}\ \mathbf{F}\ enabled$

   In all future positions, there is a future position where *enabled* holds.
6. $\mathbf{F}\ \mathbf{G}\ enabled$

   There is a future position, from which all future positions have *enabled* holding.

A Taste of LTL – Examples

1. $\mathbf{G}\ invariant$

   invariant is true for all future positions
2. $\mathbf{G}\ \neg(read \wedge write)$

   In all future positions, it is not the case that *read* and *write*
3. $\mathbf{G}(request \rightarrow \mathbf{F} grant)$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.
4. $\mathbf{G}(request \rightarrow (request\ \mathbf{U}\ grant))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.
5. $\mathbf{G}\ \mathbf{F}\ enabled$

   In all future positions, there is a future position where *enabled* holds.
6. $\mathbf{F}\ \mathbf{G}\ enabled$

   There is a future position, from which all future positions have *enabled* holding.

Computation Tree Logic (CTL)

- CTL properties are evaluated over trees.
- Every temporal operator (F, G, X, U) preceded by a path quantifier (A or E).
- Universal modalities (AF, AG, AX, AU): the temporal formula is true in **all** the paths starting in the current state.
- Existential modalities (EF, EG, EX, EU): the temporal formula is true in **some** of the paths starting in the current state.

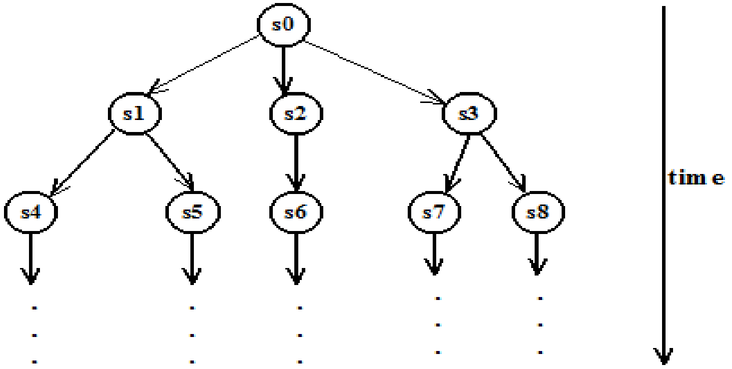
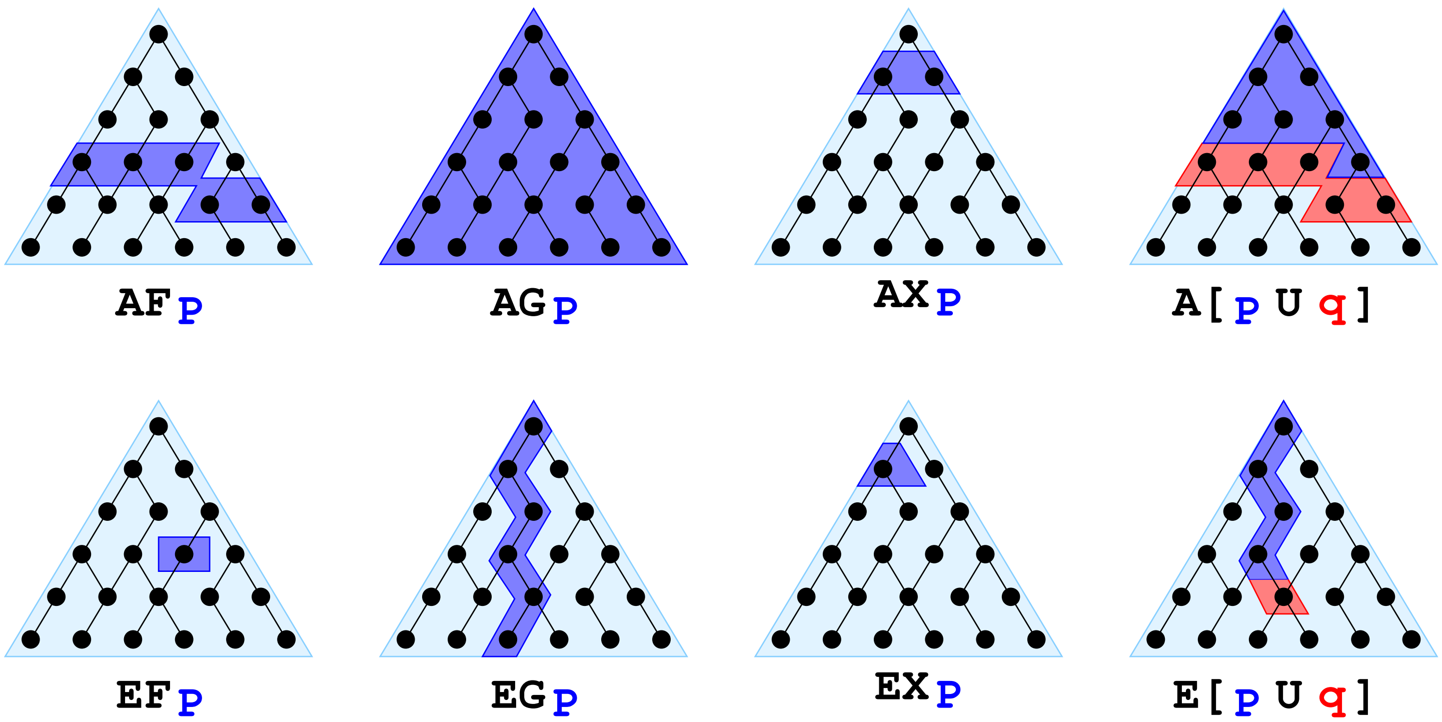


Figure 1. "Branching" progress of time

CTL

finally p | globally p | next p | p until q



CTL

- Some dualities:

$$
\begin{aligned}
AGp &\leftrightarrow \neg EF\neg p \\
AFp &\leftrightarrow \neg EG\neg p \\
AXp &\leftrightarrow \neg EX\neg p
\end{aligned}
$$

- Example: specifications for the mutual exclusion problem.

$AG\neg(C_1 \wedge C_2)$	mutual exclusion
$AG(T_1 \rightarrow AF\, C_1)$	liveness
$AG(N_1 \rightarrow EX\, T_1)$	non-blocking

C: critical, T: trying, N: non-critical

Quiz

- **Which of the following is true?**

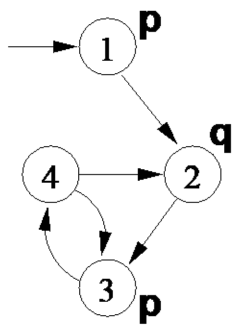
- **A: LTL $\subset$ CTL**
- **B: LTL $\supset$ CTL**
- **C: LTL $\cap$ CTL = $\emptyset$**
- **D: LTL = CTL**
- **E: LTL $\cap$ CTL $\neq$ LTL and LTL $\cap$ CTL $\neq$ CTL**
- **F: refuse to answer**

Model checking: algorithms

Model Checking

Model Checking is a formal verification technique where...

- ...the system is represented as Finite State Machine



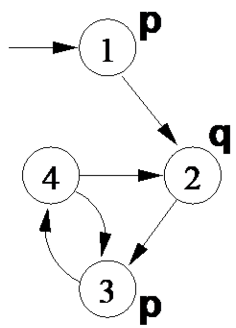
- ...the properties are expressed as temporal logic formulae

LTL: **G(p –> Fq)** CTL: **AG(p –> AFq)**

- ...the model checking algorithm checks whether all the executions of the model satisfy the formula.

CTL Model Checking: Example

Consider a simple system and a specification:

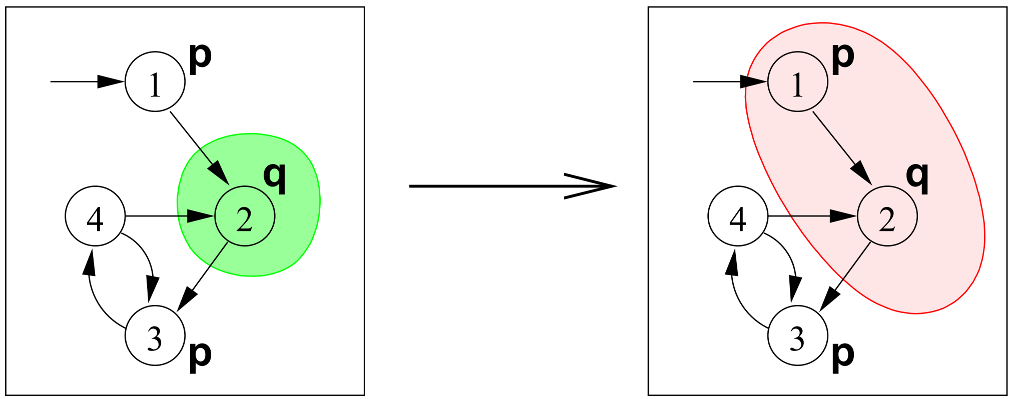


AG(p –> AFq)

Idea:

- construct the set of states where the formula holds
- proceeding "bottom-up" on the structure of the formula
- **q, AFq, p, p $\to$ AF q, AG(p $\to$ AF q)**

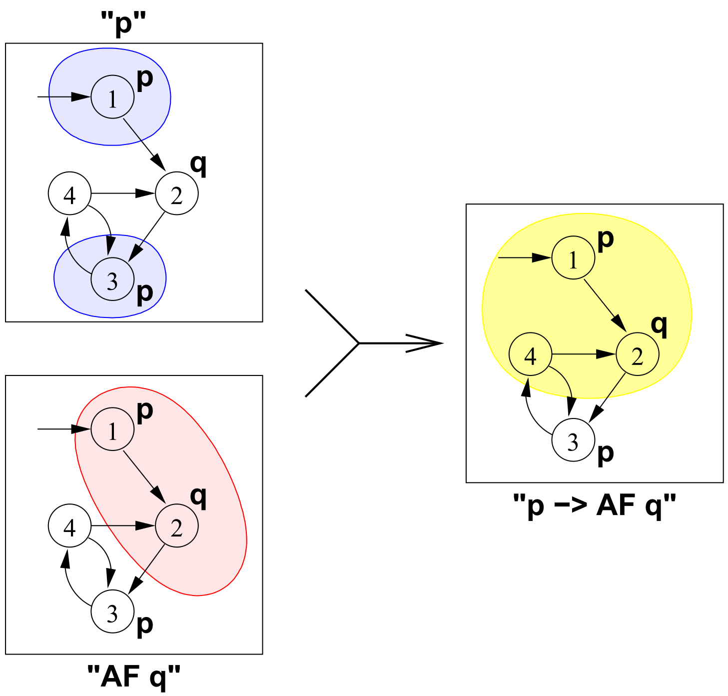
CTL Model Checking: Example



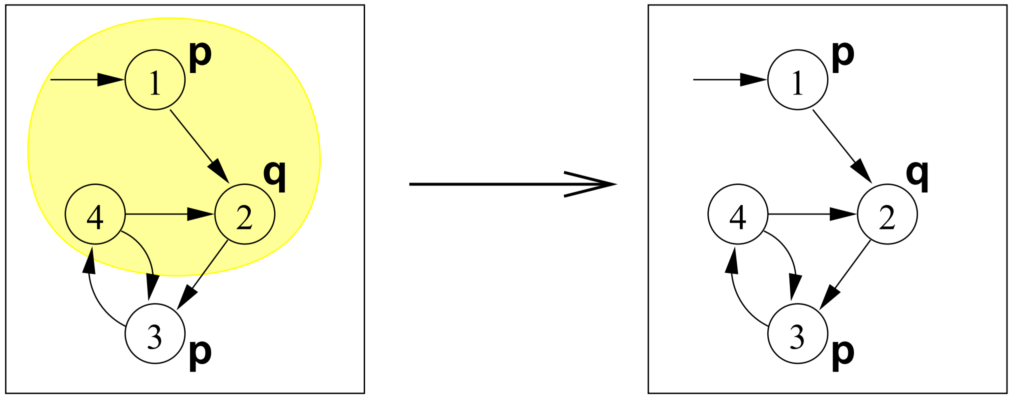
"q" **"AF q"**

AF q is the union of **q**, **AX q**, **AX AX q**, ...

CTL Model Checking: Example



CTL Model Checking: Example



"p −> AF q" **"AG(p −> AF q)"**

The set of states where the formula holds is empty!

Counterexample reconstruction is based on the intermediate sets.

The Main Problem: State Space Explosion

The bottleneck:

- Exhaustive analysis may require to store all the states of the Kripke structure
- The state space may be exponential in the number of components
- State Space Explosion: too much memory required

Symbolic Model Checking:

- Symbolic representation
- Different search algorithms

The NuSMV Model Checker

NuSMV

NuSMV provides:

1. A language for describing finite state models of systems
   - Reasonably expressive
   - Allows for modular construction of models
2. Model checking algorithms for checking specifications written in LTL and CTL (and some other logics) against finite state machines.

A first SMV program

```
MODULE main
  VAR
    b0 : boolean
  ASSIGN
    init(b0) := FALSE;
    next(b0) := !b0;
```

An SMV program consists of:

- Declarations of state variables (`b0` in the example); these determine the state space of the model.
- Assignments that constrain the valid initial states (`init(b0) := FALSE`).
- Assignments that constrain the transition relation (`next(b0) := !b0`).

Declaring state variables

SMV data types include:

boolean:

```
x : boolean;
```

enumeration:

```
st : {ready, busy, waiting, stopped};
```

bounded integers (intervals):

```
n : 1..8;
```

arrays and bit-vectors

```
arr : array 0..3 of {red, green, blue};
bv  : signed word[8];
```

Assignments

initialisation:

```
ASSIGN
init(x) := expression ;
```

progression:

```
ASSIGN
next(x) := expression ;
```

immediate:

```
ASSIGN
y := expression ;
```

or

```
DEFINE
y := expression ;
```

Assignments

- If no **init()** assignment is specified for a variable, then it is initialised non-deterministically;
- If no **next()** assignment is specified, then it evolves nondeterministically. i.e. it is unconstrained.
  - Unconstrained variables can be used to model nondeterministic inputs to the system.
- Immediate assignments constrain the current value of a variable in terms of the current values of other variables.
  - Immediate assignments can be used to model outputs of the system.

Expressions

$$
\begin{array}{rcll}
\mathit{expr} & ::= & \text{atom} & \text{symbolic constant} \\
& | & \text{number} & \text{numeric constant} \\
& | & \text{id} & \text{variable identifier} \\
& | & \texttt{!}\,\mathit{expr} & \text{logical not} \\
& | & \mathit{expr} \bowtie \mathit{expr} & \text{binary operation} \\
& | & \mathit{expr}\texttt{[}\mathit{expr}\texttt{]} & \text{array lookup} \\
& | & \texttt{next}(\mathit{expr}) & \text{next value} \\
& | & \mathit{case\_expr} & \\
& | & \mathit{set\_expr} &
\end{array}
$$

where $\bowtie \in \{\texttt{\&}, \texttt{|}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{=}, \texttt{!=}, \texttt{<}, \texttt{<=}, \ldots\}$

Case Expression

$$
\begin{array}{l}
\textit{case\_expr} ::= \\
\quad \texttt{case} \\
\qquad \textit{expr}_{a1} \;\; \texttt{:} \;\; \textit{expr}_{b1}\,\texttt{;} \\
\qquad \ldots \\
\qquad \textit{expr}_{an} \;\; \texttt{:} \;\; \textit{expr}_{bn}\,\texttt{;} \\
\quad \texttt{esac}
\end{array}
$$

- Guards are evaluated sequentially.
- The first true guard determines the resulting value

Set expressions

Expressions in SMV do not necessarily evaluate to one value.

- In general, they can represent a set of possible values.
  `init(var) := {a,b,c} union {x,y,z} ;`
- destination (lhs) can take any value in the set represented by the set expression (rhs)
- constant `c` is a syntactic abbreviation for singleton `{c}`

LTL Specifications

- LTL properties are specified with the keyword `LTLSPEC`:
  `LTLSPEC <ltl_expression> ;`
- `<ltl_expression>` can contain the temporal operators:
  `X_ F_ G_ _U_`
- E.g. condition `out = 0` holds until `reset` becomes false:
  `LTLSPEC (out = 0) U (!reset)`

ATM Example

```
MODULE main
VAR
  state: {welcome, enterPin, tryAgain, askAmount,
          thanksGoodbye, sorry};
  action: {cardIn, correctPin, wrongPin, ack, cancel,
           fundsOK, problem, none};
ASSIGN
  init(state) := welcome;
  next(state) := case
    state = welcome & action = cardIn        : enterPin;
    state = enterPin & action = correctPin : askAmount ;
    state = enterPin & action = wrongPin   : tryAgain;
    state = tryAgain & action = ack        : enterPin;
    state = askAmount & action = fundsOK   : thanksGoodbye;
    state = askAmount & action = problem   : sorry;
    state = enterPin & action = cancel     : thanksGoodbye;
    TRUE                                   : state;
  esac;
LTLSPEC F( G state = thanksGoodbye
          | G state = sorry
         );
```

Running NuSMV

Batch

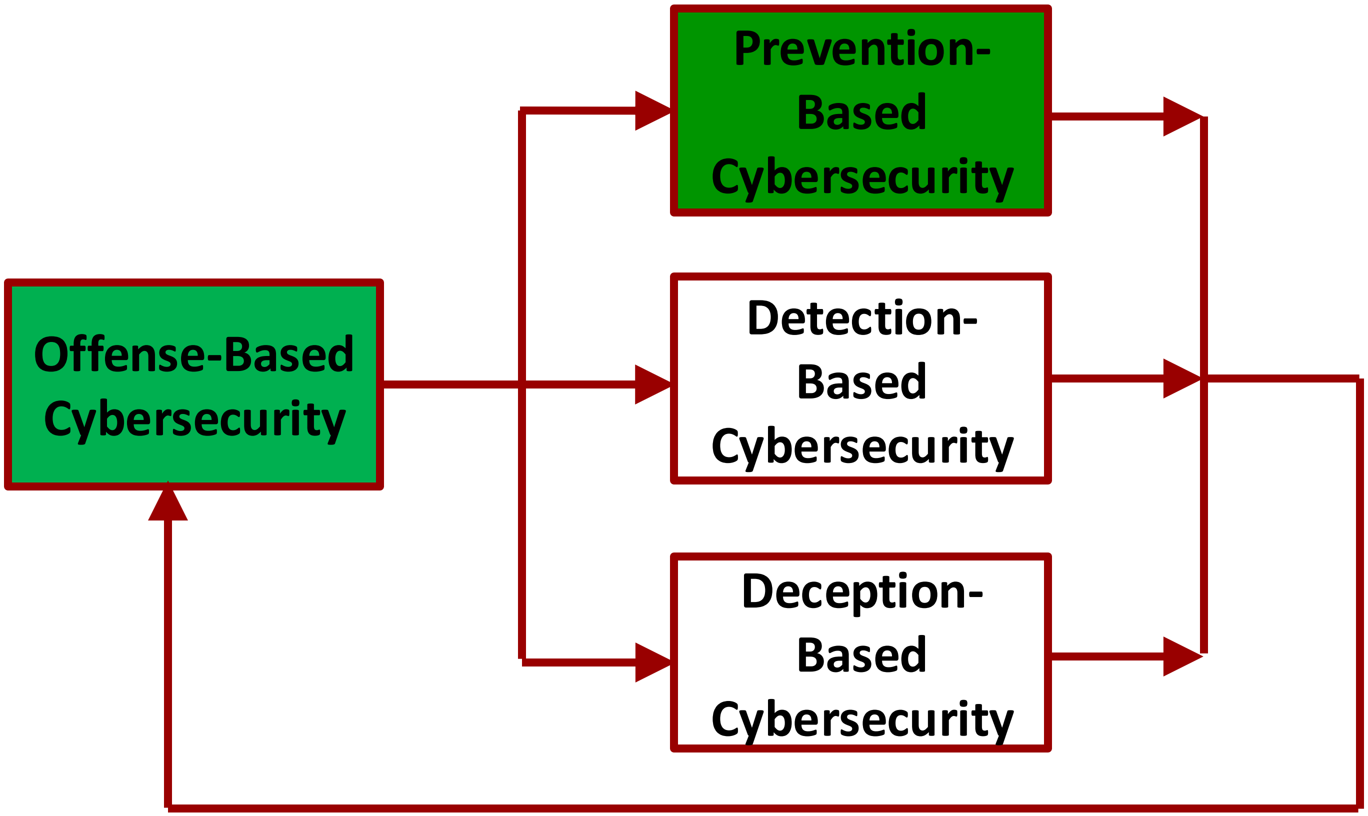
```
$ NuSMV atm.smv
```

Interactive

```
$ NuSMV -int atm.smv
NuSMV > go
NuSMV > check_ltlspec
NuSMV > quit
```

- go abbreviates the sequence of commands `read_model`, `flatten_hierarchy`, `encode_variables`, `build_model`.
- For command options, use `-h` or look in the NuSMV User Manual.

Course structure



End of Lecture 13