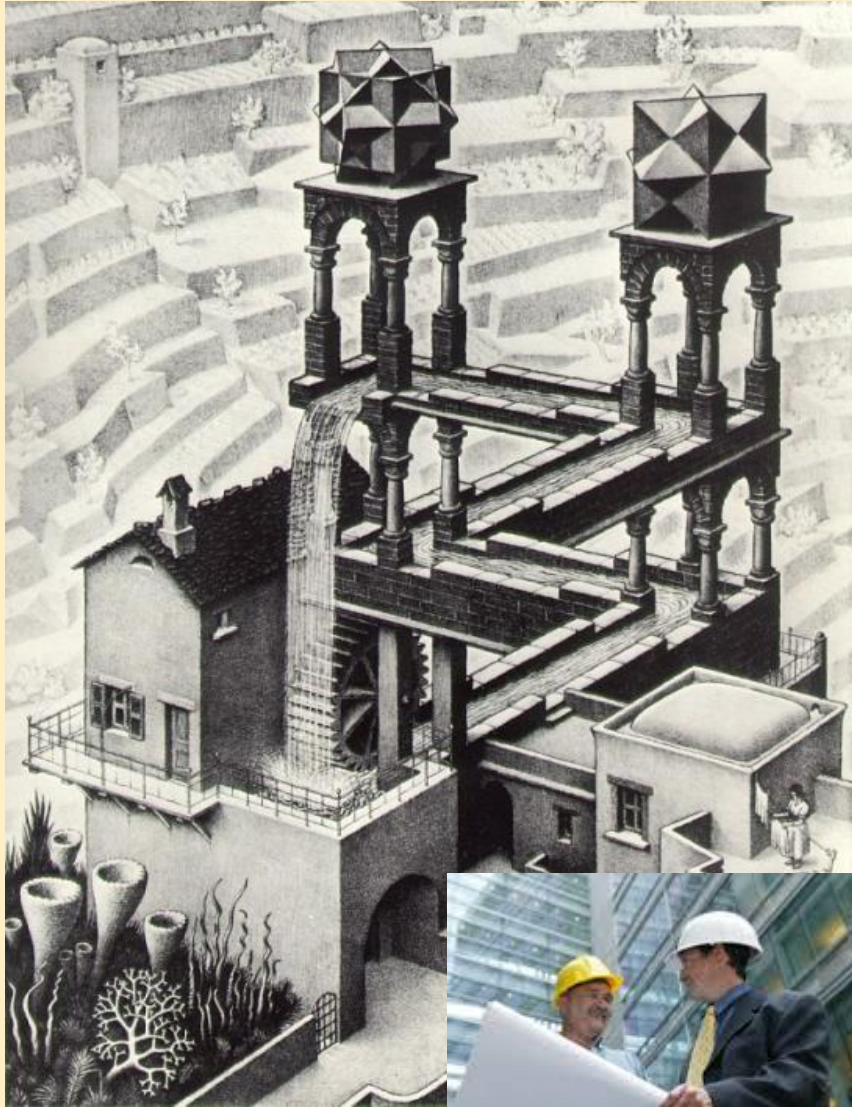


Formalizing Requirements: Temporal Logics

dr. István Majzik

BME Department of Measurement and Information Systems



```

DSE CTS
*** STOP: 0x00000000 (0x00000000, 0x0000001a, 0x00000000, 0x00000000)
IRQL_NOT_LESS_OR_EQUAL

p4-0300 irq1:1f SYSVER:0xf000030e

Dll Base DateStmp - Name                               Dll Base DateStmp - Name
80100000 2a53e338 - ntosknl.exe                               80400000 2a53eb38 - hal.dll
80010000 2a41884b - Ahal54w.sys                               80013000 2a4b229a - SCSSIPORT.SYS
8001b000 2a4e7b6b - ScsiDisk.sys                               80220000 2a53f238 - Ntfs.sys
Fe420000 2a406607 - Floppy.SYS                               fe430000 2a406618 - ScsiCdm.SYS
Fe440000 2a406659 - Fs_Rcd.SYS                               fe450000 2a40660f - Null.SYS
Fe460000 2a406524 - Bcap.SYS                               fe470000 2a406634 - Scrmouse.SYS
Fe480000 2a42a424 - i8042prt.SYS                               fe490000 2a4066d4 - Hwclass.SYS
Fe4a0000 2a40660c - Kbdclass.SYS                               fe4c0000 2a4065e2 - VIDEOPT.SYS
Fe4b0000 2a53d4d9 - ata.SYS                               fe4d0000 2a4065e8 - vga.sys
Fe4e0000 2a406655 - Msfs.SYS                               fe4f0000 2a414f30 - Mpfs.SYS
Fe510000 2a53f222 - MDIS.SYS                               fe500000 2a40719b - clnkii.sys
Fe550000 2a406697 - TDI.SYS                               fe530000 2a47c740 - nbfi.sys
Fe560000 2a527949 - nwlankipx.sys                               fe570000 2a53a89e - nwlankmb.sys
Fe580000 2a494973 - tcpip.sys                               fe5a0000 2a525688 - afd.sys
Fe5b0000 2a527943 - netbt.sys                               fe5d0000 2a4167f7 - netbios.sys
Fe5e0000 2a4066b3 - mup.sys                               fe5f0000 2a4f9f51 - rdr.sys
Fe630000 2a53f24a - srv.sys                               fe660000 2af16062 - nwlanksp.sys

Address: dword dump Build [1057]
FF541B4c Fe5105df fe5105df 00000001 ff540128 fe4a8228 000002fe - Name
ff541a60 fe501368 fe501368 00000246 00004002 00000000 00000000 - clnkii.sys
ff541eb4 fe481509 fe481509 ff6688c8 ff668238 00000000 ff668138 - i8042prt.SYS
ff541a00 fe481a38 fe481a38 fe482078 00000000 ff541f04 8013c58a - i8042prt.SYS
ff541a04 fe482078 fe482078 00000000 ff541f04 8013c58a ff6688c8 - i8042prt.SYS
ff541af0 8013c58a 8013c58a ff6688c8 ff668040 80405900 00000031 - ntosknl.exe
ff541afc 80405900 80405900 00000031 06060606 06060606 06060606 - hal.dll

Restart and set the recovery options in the system control panel
or the /CRASHDEBUG system start option if this message reappears,
contact your system administrator or technical support group.
CRASHDUMP: Initializing minidump driver
CRASHDUMP: Dumping physical memory to disk: 2000
CRASHDUMP: Physical memory dump complete

```

```

C:\WINDOWS\system32\cmd.exe

C:\myworkspace>javac numerator1.java
C:\myworkspace>java numerator1

Enter the Numeric:
123
You have Entered:123

java.lang.Exception: Error at:Fri Dec 03 22:03:04 AMT 2010
at numerator1.MyNumerator(Numerat
at numerator1.main(Numerator1.Jav
Caused by: java.lang.ArithmeticException:
... 1 more

C:\myworkspace>

```



What is the point of formalizing requirements?

Motivating example: mutual exclusion

- 2 processes, 3 shared variables (H. Hyman, 1966)
 - **blocked0**: process 1 (P0) wants to enter
 - **blocked1**: process 2 (P1) wants to enter
 - **turn**: which process is allowed to enter (0 for P0, 1 for P1)

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

P0

```
while (true) {
    blocked1 = true;
    while (turn!=1) {
        while (blocked0==true) {
            skip;
        }
        turn=1;
    }
    // Critical section
    blocked1 = false;
    // Do other things
}
```

P1

Is the algorithm correct?

The model in UPPAAL (version 1)

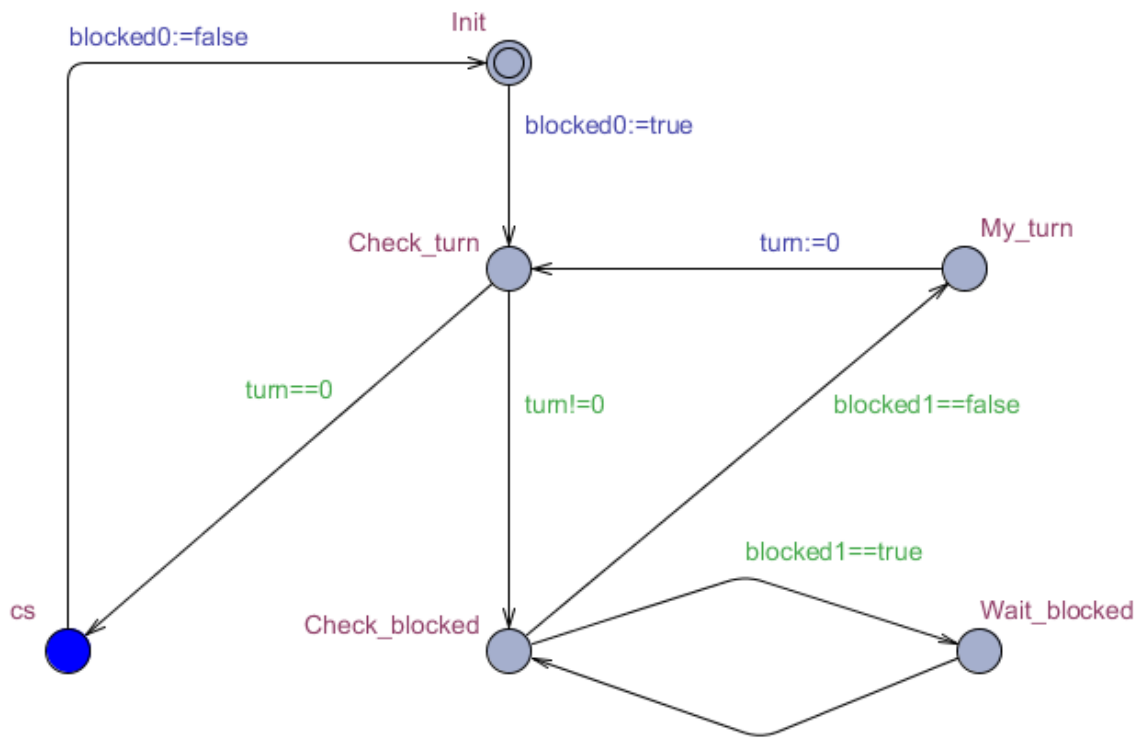
Declarations:

```
bool blocked0;  
bool blocked1;  
int[0,1] turn=0;  
system P0, P1;
```

Modeling idioms used:

- Global variables
- Variables with restricted domain

Automaton P0:



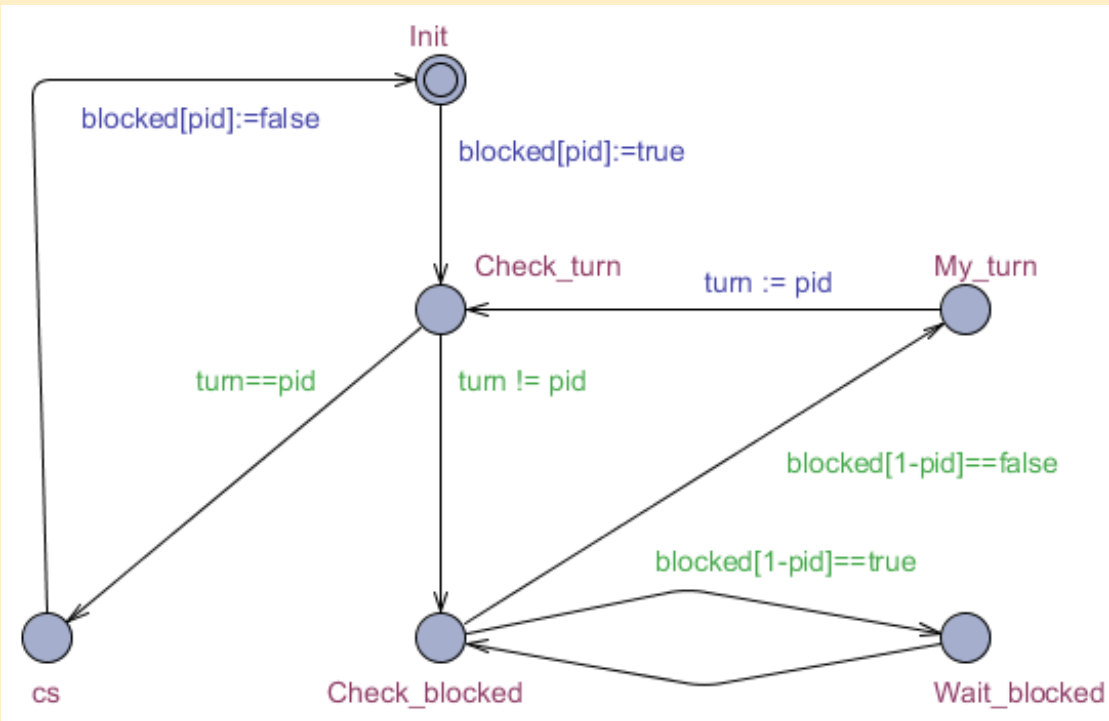
```
while (true) {                                P0  
  blocked0 = true;  
  while (turn!=0) {  
    while (blocked1==true) {  
      skip;  
    }  
    turn=0;  
  }  
  // Critical section  
  blocked0 = false;  
  // Do other things  
}
```

The model in UPPAAL (version 2)

Declarations:

```
bool blocked[2];  
int[0,1] turn;  
P0 = P(0);  
P1 = P(1);  
system P0,P1;
```

Template P with parameter pid:



Modeling idioms used:

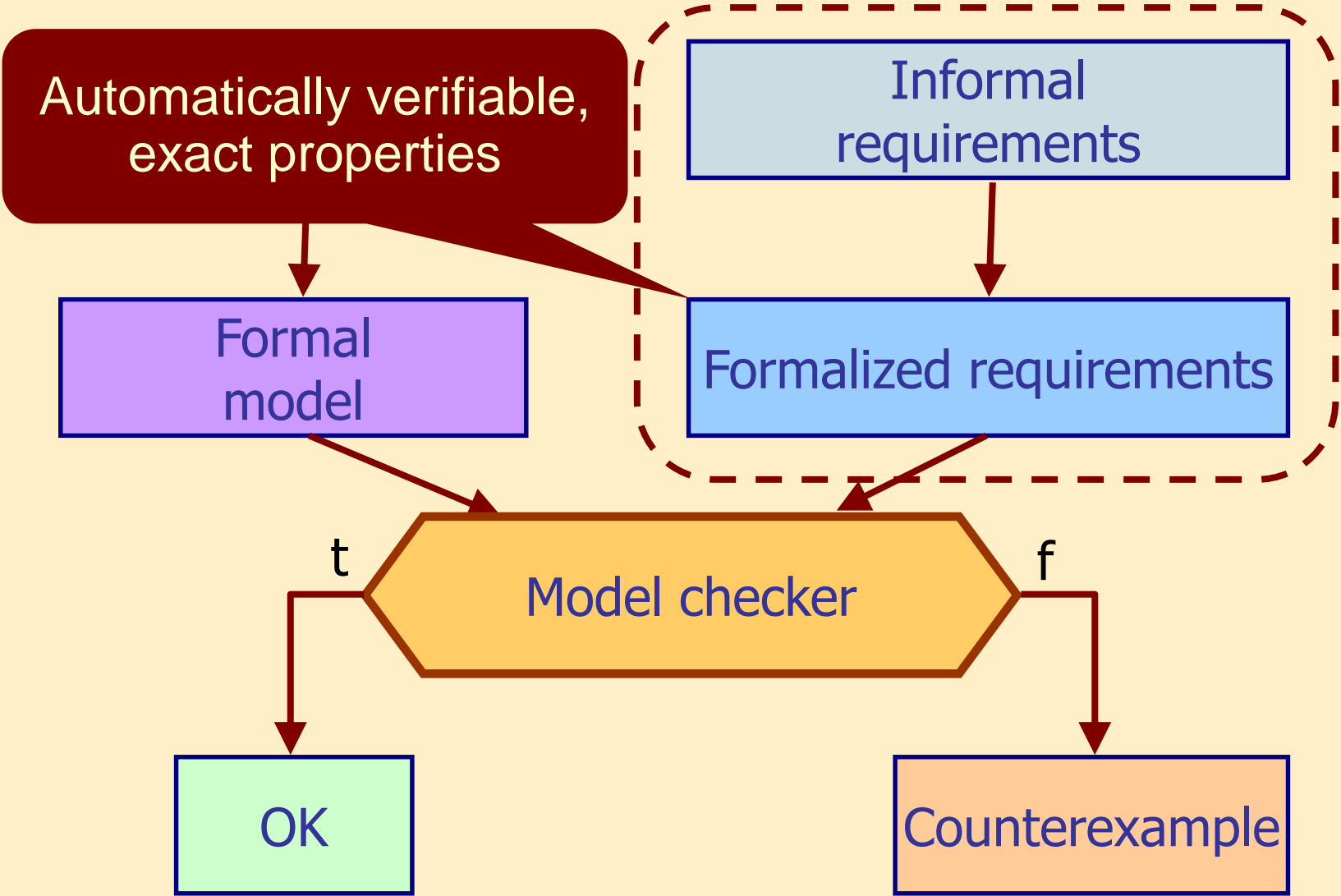
- Global variables
- Variables with restricted domain
- Modeling common behavior with templates
- Template instantiation with parameters
- Variables of array type

```
while (true) { P0  
  blocked0 = true;  
  while (turn!=0) {  
    while (blocked1==true) {  
      skip;  
    }  
    turn=0;  
  }  
  // Critical section  
  blocked0 = false;  
  // Do other things  
}
```

Properties to verify

- Mutual exclusion:
 - At most one process is allowed to be in the critical section
- The expected behavior is possible:
 - For P0 it is possible to enter the critical section
 - For P1 it is possible to enter the critical section
- Starvation freedom:
 - P0 will eventually enter the critical section
 - P1 will eventually enter the critical section
- Deadlock freedom:
 - It is not possible that processes are mutually waiting for each other

Our goal



Establishing and formalizing requirements

What are the typical requirements
(in critical systems)?

What to formalize?

Handling textual requirements

- Specifying a typical requirement: text

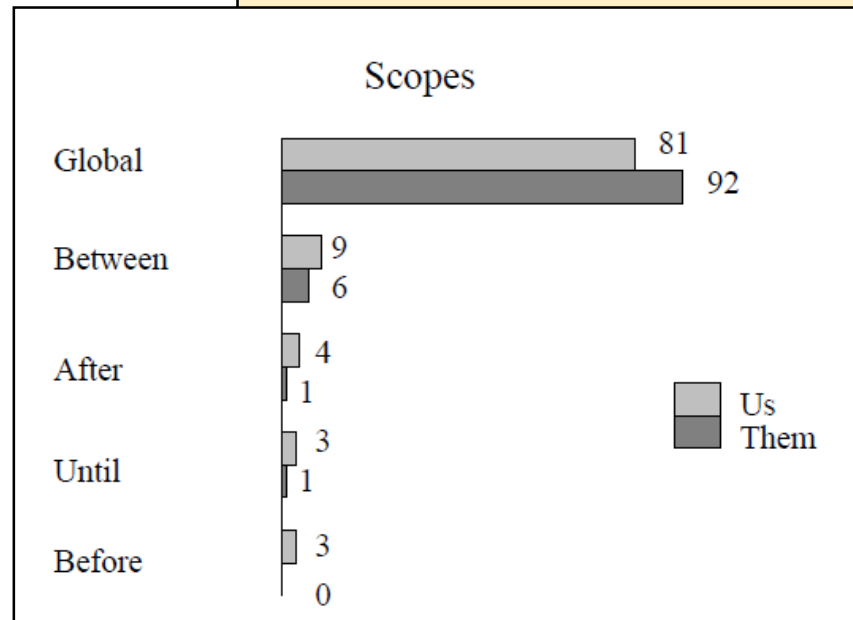
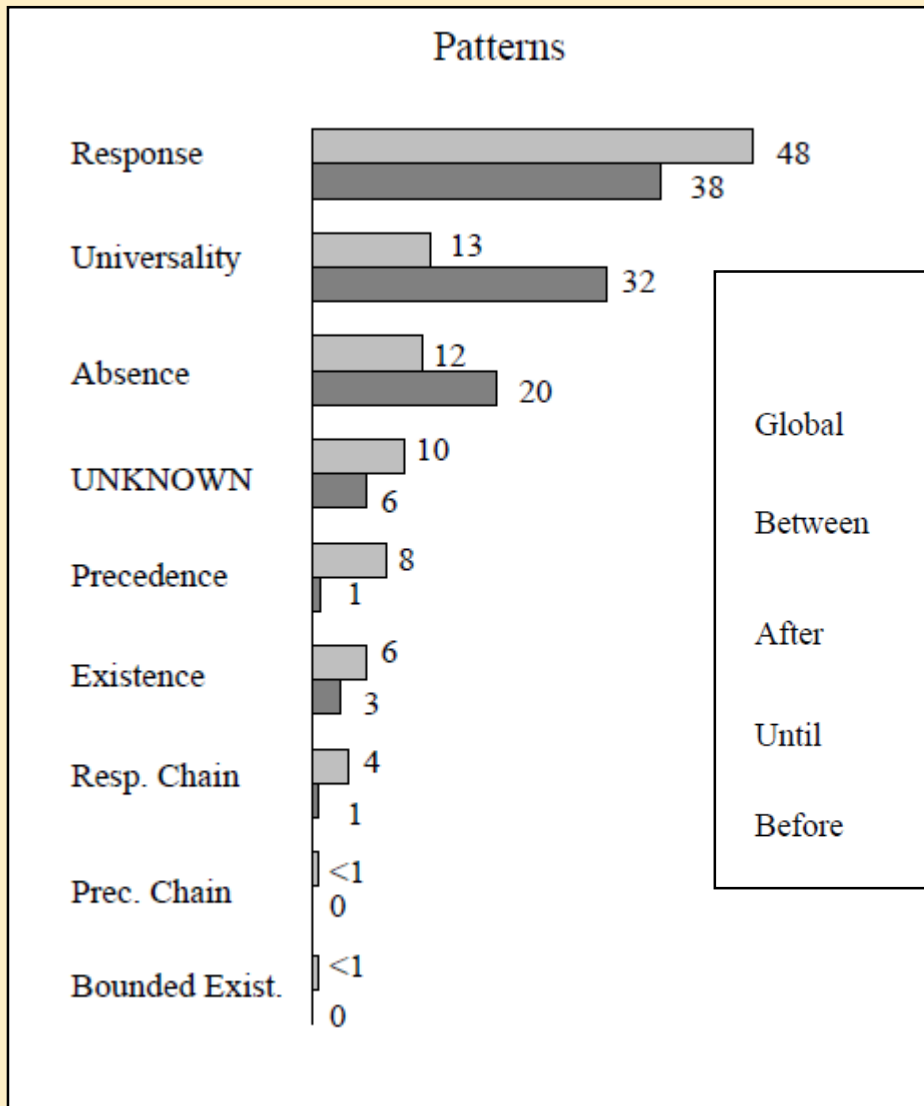
If alarm is on and alert occurs, the output of safety should be true as long as alarm is on.

If the switch is turned to AUTO, and the light intensity is LOW then the headlights should stay or turn immediately ON, afterwards the headlights should continue to stay ON in AUTO as long as the light intensity is not HIGH.

- Is the textual description unambiguous?
- Structure is not clear
(condition, requirement, output, timing, ...)

The result of a survey

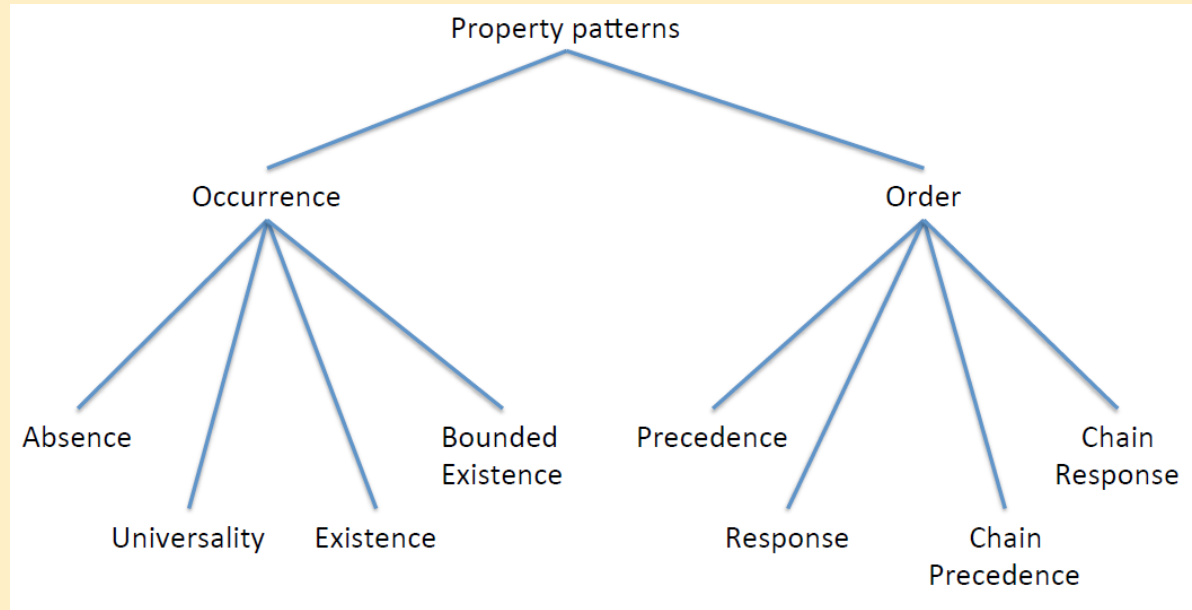
A significant proportion of requirements match to certain patterns



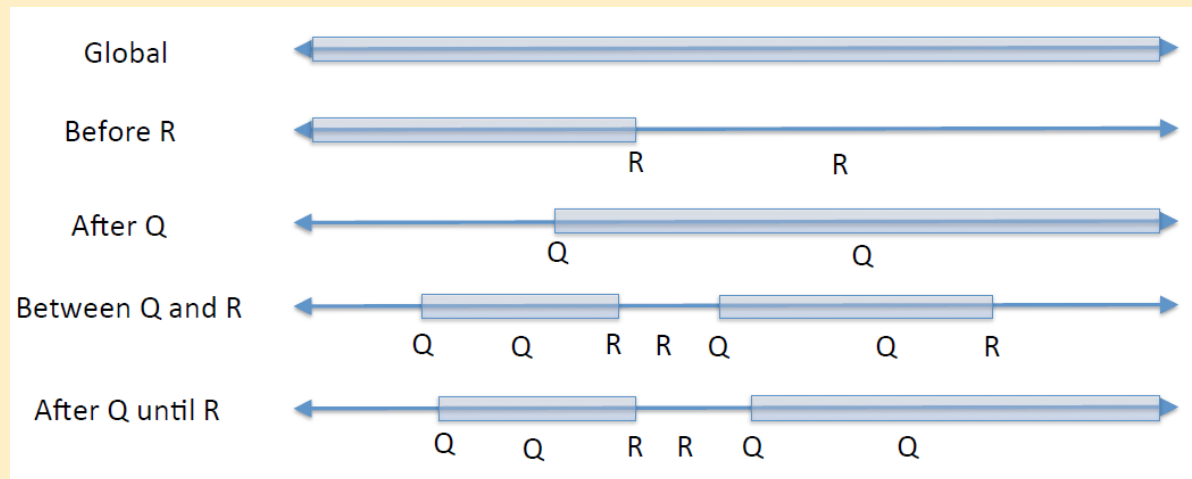
Figures: The distribution of matched patterns for requirements from two development teams

Groups of patterns

Pattern:
order or
occurrence



Scope:
relative to
further events



Patterns

Occurrence:

- **Absence:** the referenced state/event never occurs
- **Universality:** the referenced state/event is always present
- **Existence:** the referenced state/event eventually occurs
- **Bounded existence:** the referenced state/event occurs at least k times

Order:

- **Precedence:** the referenced state/event precedes an other state/event
- **Response:** the referenced state/event is proceeded by an other state/event
- **Chain precedence:** generalization of Precedence to sequences
- **Chain response:** generalization of Response to sequences

Examples of patterns

- **Pattern Response in scope Global:**

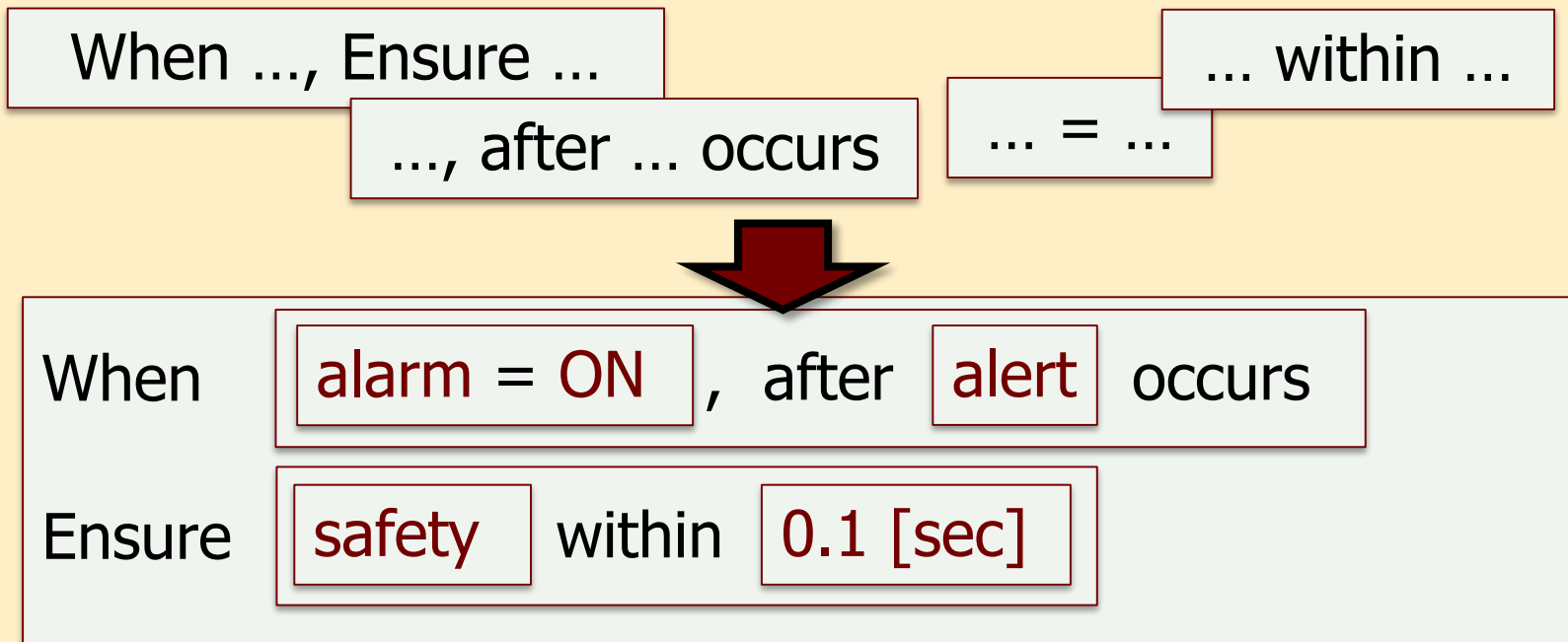
At any time during execution,
if event **Request** occurs,
then it should be proceeded by either **Reply** or **Reject**.

- **Pattern Precedence in scope After:**

After the occurrence of state **NormalMode**,
state **ResourceGranted** may only occur
if it is preceded by state **ResourceRequest**.

A typical solution

- The use of textual templates*
 - Composing parameterized patterns
 - More transparent structure
 - Formal semantics can be assigned to the patterns



Example: Semantics of a pattern

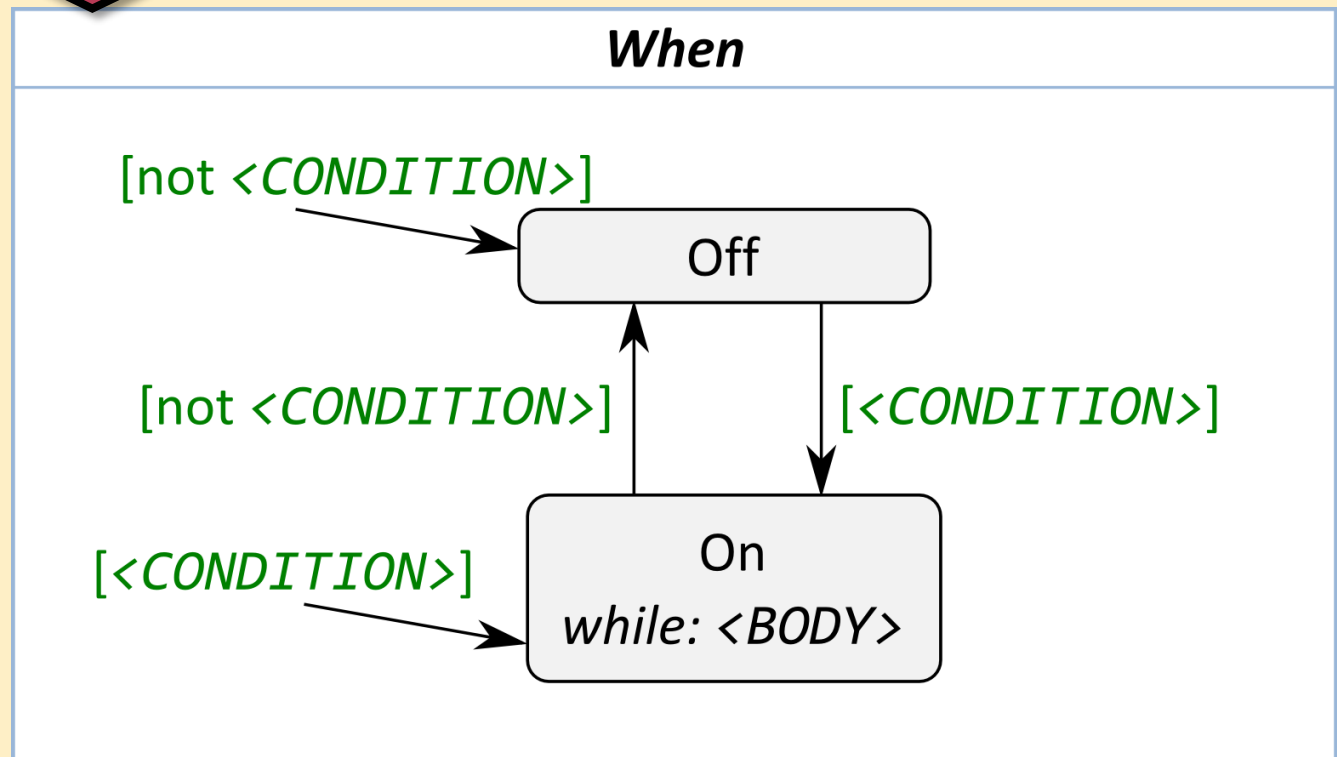
When *<CONDITION>*, Ensure *<BODY>*

=

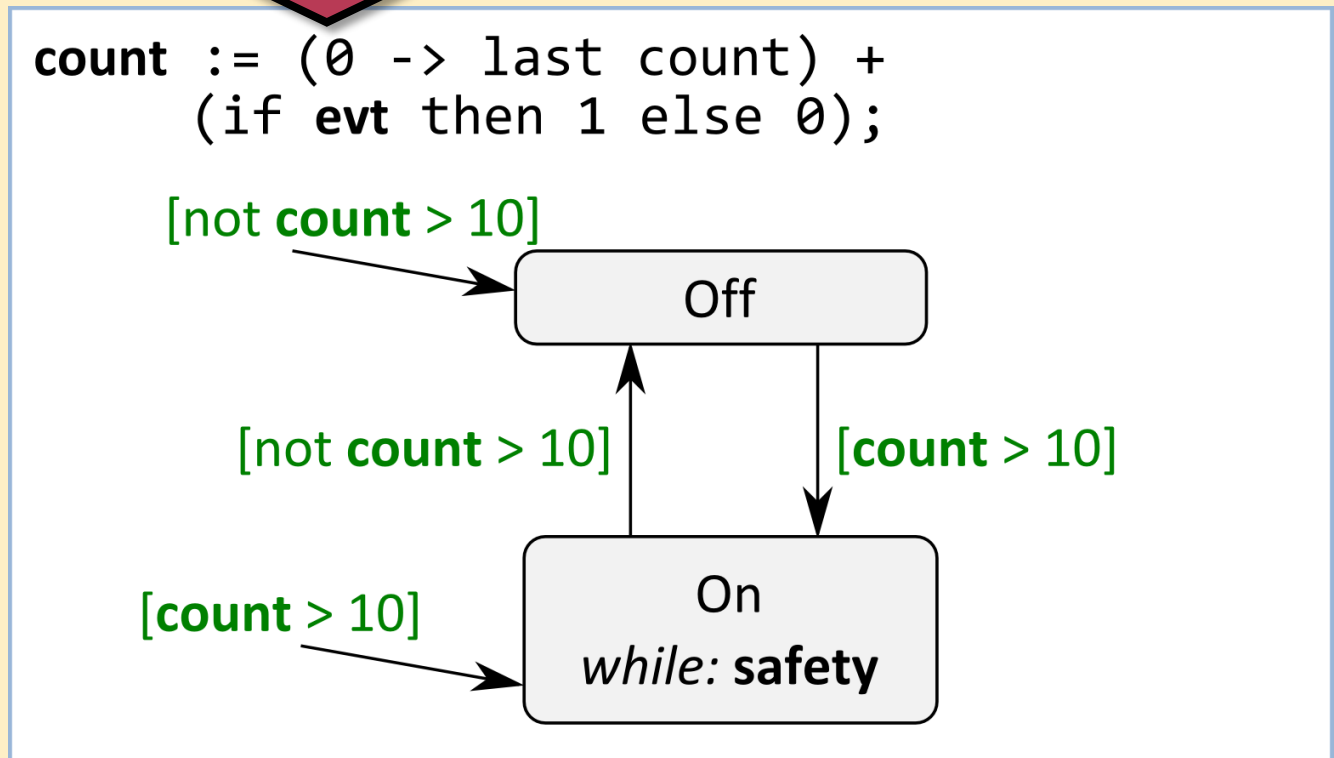
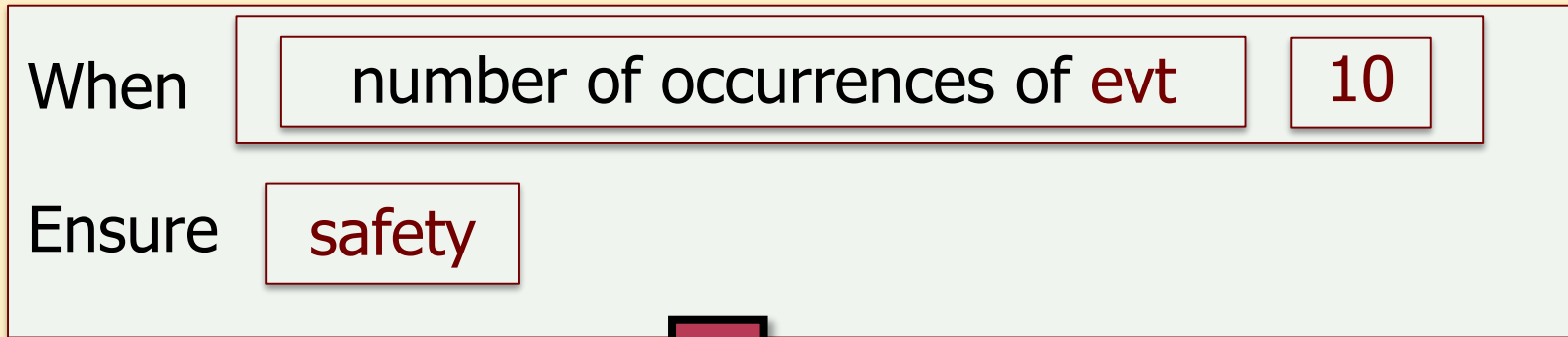
When *<CONDITION>* becomes true until it becomes false, do *<BODY>*



Here, the semantics of a block is given by a state machine



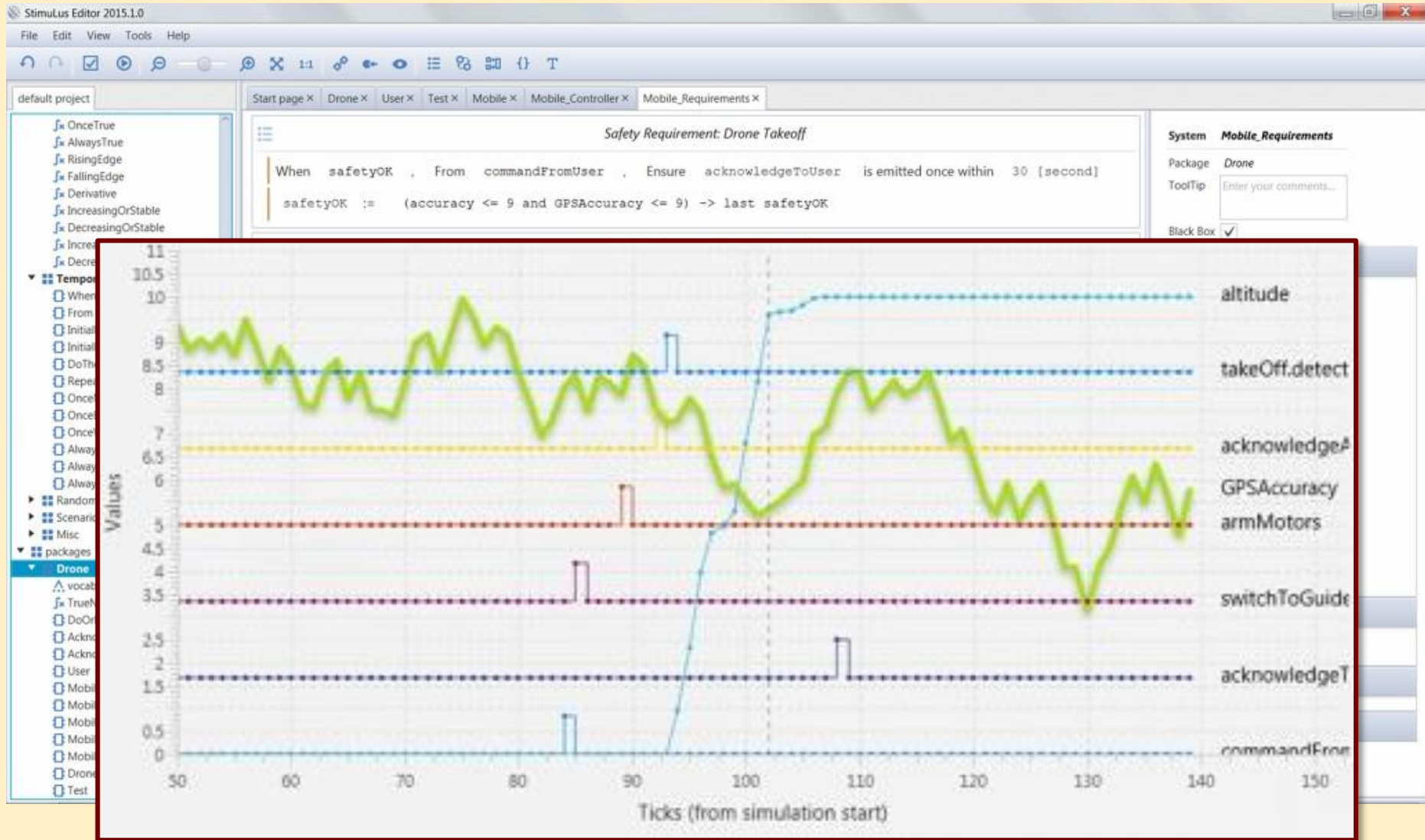
Example: Semantics of composite patterns



The use of formalized requirements

- Validation
 - Executions can be generated that satisfy the requirement
 - Executions can be evaluated w.r.t. to the requirements
 - Are we specifying what we think we do?
 - Is the set of requirements complete and unambiguous?
- Formal verification
 - Verification of design (models)
- Generating a test oracle
 - Verification of implementation (in a testing environment)
- Documentation
 - Readable, but formalized and validated

Example: Argosim Stimulus tool



The takeaways

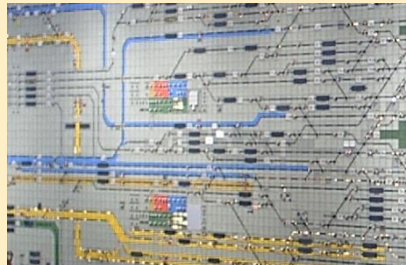
- The majority of properties match certain patterns
 - If ... then ..., While ... ensure ..., After ...
 - Occurrence/order of states/events
- More complex requirements can be composed from simpler ones
 - Parametrization: properties of a state/event
 - Nesting
- Formalization of requirements helps
 - Analysis of requirements: validity, completeness, consistency
 - Verification of design: exhaustive analysis of executions
 - Test evaluation, runtime monitoring: components can be automatically generated

Temporal requirements

What kind of requirements do we formalize?

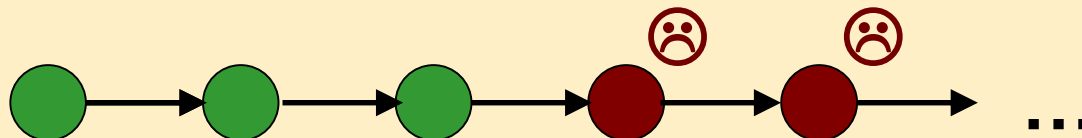
- Verification: Model \leftrightarrow Many requirements
 - Functional: logically correct behavior <- our current goal
 - Extra-functional: performance, reliability, ... <- later
- Goal: verifying reachability of states
 - System (model): we know local properties of states
 - Name, valuation of variables, mode of operation, ...
 - Requirements: order of occurrence of states
 - Is a desirable state reachable? -> Liveness properties
 - Are we avoiding dangerous states? -> Safety properties

Can be verified by exhaustive exploration of the state space!
- Important in state based, event driven systems



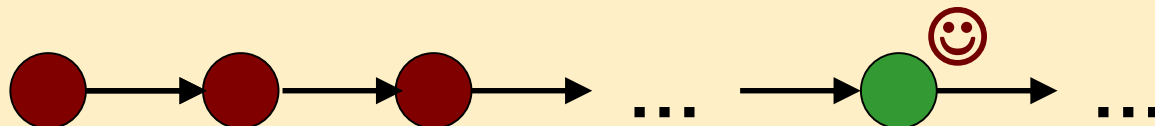
Safety properties

- Expresses freeness from dangerous situations
 - “In all states, the pressure is below the critical level”
 - “The press machine only operates with closed barriers.”
- Examples from Computer Science:
 - Deadlock freedom: no deadlock can occur
 - Mutual exclusion: at most one process in the critical section
 - Data confidentiality: no unauthorized accesses
- Universal property on reachable states:
 - “In all reachable states it holds that ...”
 - Formulates an invariant
- If a sequence of states violates it:
then already a finite prefix of the sequence violates it



Liveness properties

- Expresses reachability of a **desirable state**
 - “After start the press machine emits the finished product.”
 - “After the disturbance the system stabilizes.”
- Example from Computer Science:
 - “The process gets served”
 - “The sent message arrives”
 - “The process provides the expected result on its output”
- **Existential property on reachable states**
 - “There exists a reachable state such that ...”
 - Formulates occurrence
- If a sequence of states violates it:
then it can be extended so that it satisfies the property



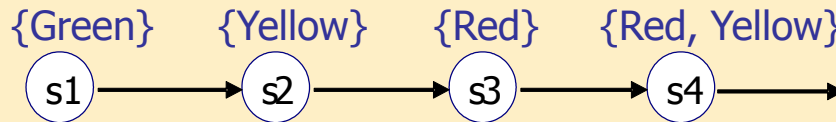
What kind of description language is needed?

- **Reachability: occurrence and order of states**
 - **Order: logical time**
 - Current point in time: current state
 - Subsequent points in time: next state(s)
 - **Temporal connectives can be used to express requirements**
- **Temporal logics:**
 - **Formal system for evaluating changes in logical time**
 - **Temporal connectives:**
"always", "at some point", "before", "while" ...
(correspond to typical requirement patterns)

Classification of temporal logics

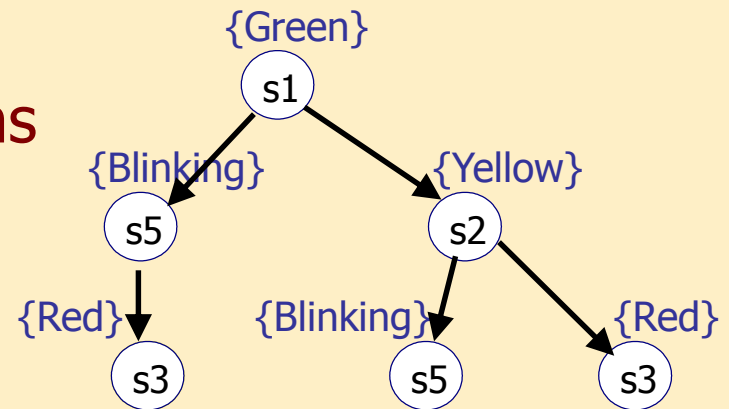
- **Linear:**

- We consider individual executions of the system
- Each state has exactly one subsequent state
- Logical time along a linear timeline (trace)



- **Branching:**

- We consider trees of executions of the system
- Each state possibly has many subsequent state
- Logical time along a branching timeline (computation tree)



Temporal logics

Where can we use temporal logics?

- Goal: examining the state space

The simplest mathematical model: Kripke structure

- We express local properties of states by **labeling**

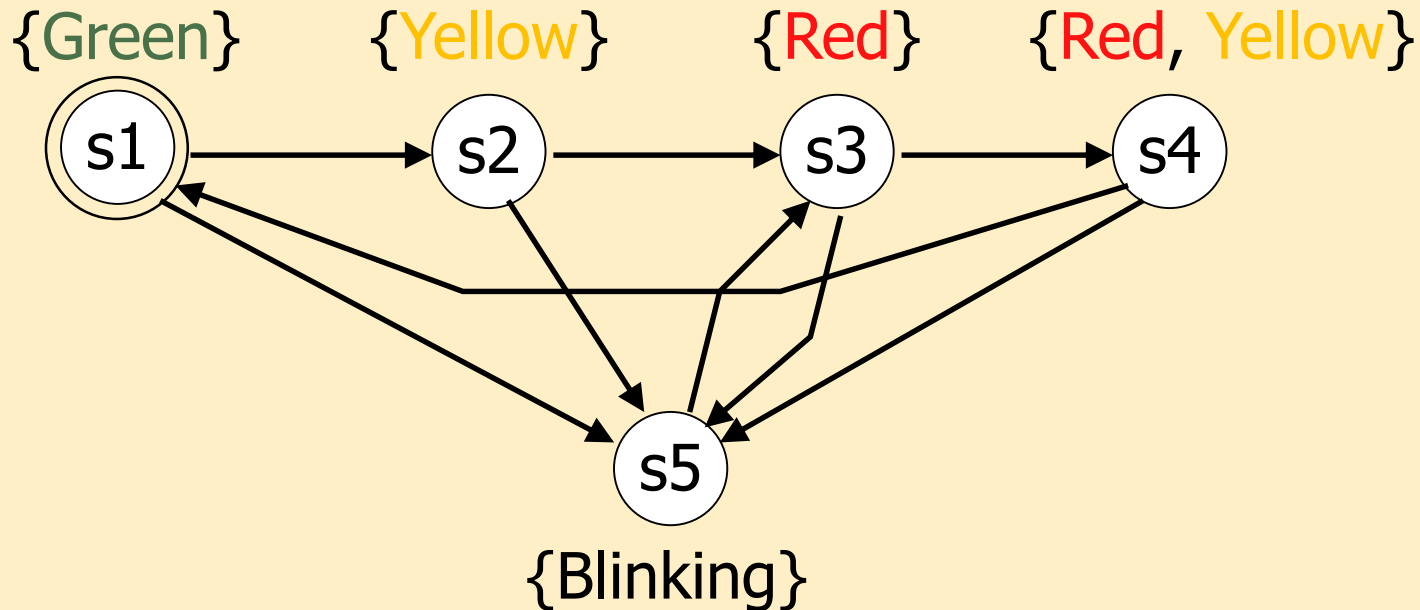
A Kripke structure KS over a set of atomic propositions $AP = \{P, Q, R, \dots\}$ is a tuple (S, I, R, L) where

- $S = \{s_1, s_2, \dots, s_n\}$ is a finite set of states,
- $I \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is the set of transitions and
- $L : S \rightarrow 2^{AP}$ is the labeling of states by atomic propositions

Example for KS

Traffic light

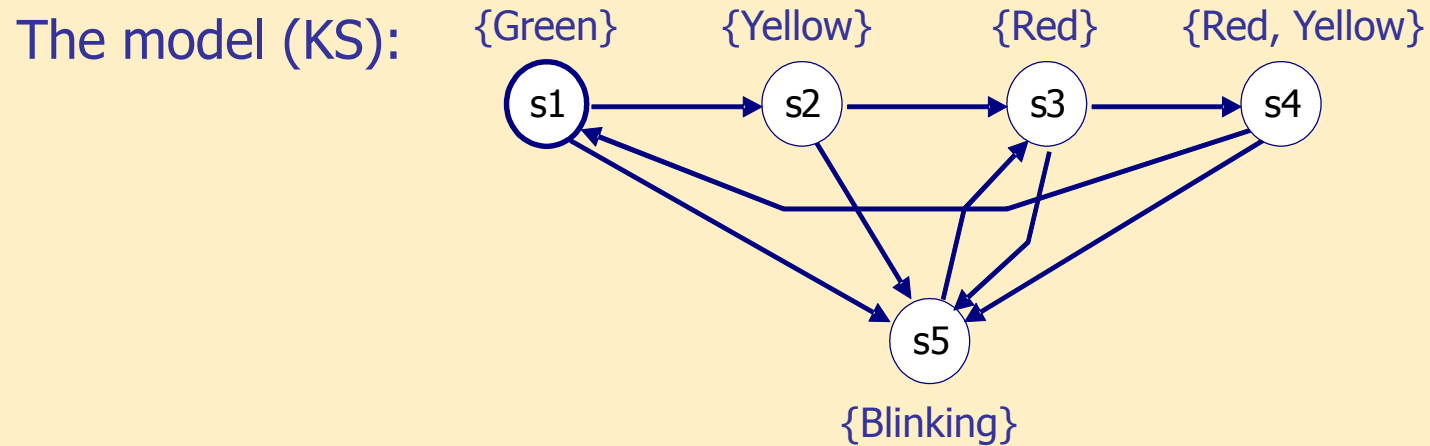
- $AP = \{\text{Green, Yellow, Red, Blinking}\}$
- $S = \{s_1, s_2, s_3, s_4, s_5\}$



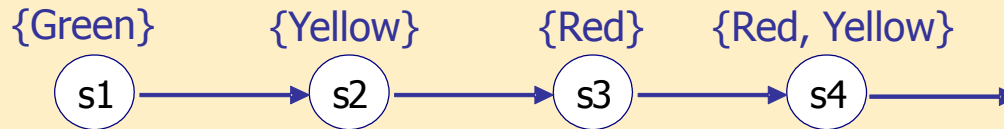
Linear Temporal Logic: LTL

Linear Temporal Logic

- Interpreted over **paths** of a Kripke structure
 - e.g. the effects of a concrete input



A path (sequence of states):



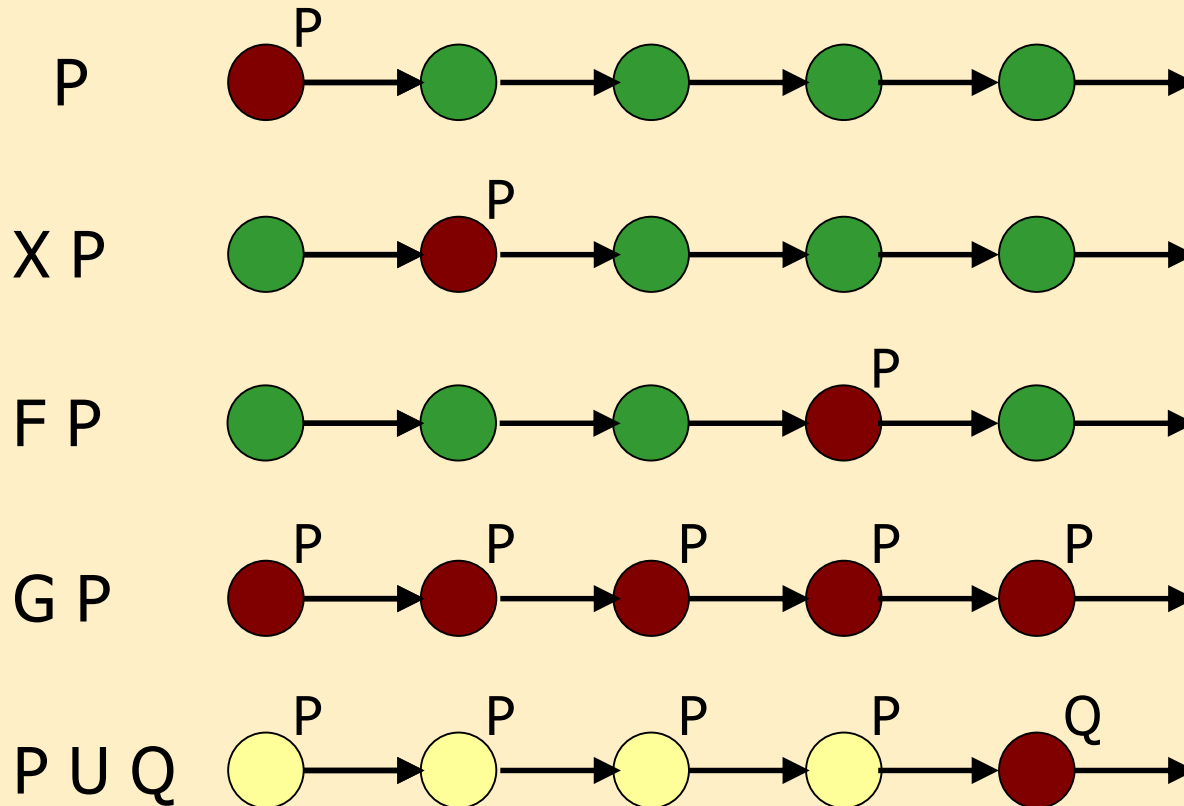
Linear temporal logic – Formulas

Construction of formulas: p, q, r, \dots

- Atomic propositions (elements of AP): P, Q, \dots
- Boolean connectives: $\wedge, \vee, \neg, \Rightarrow$
 \wedge : conjunction, \vee : disjunction, \neg : negation, \Rightarrow : implication
- Temporal connectives: X, F, G, U informally:
 - $X p$: “neXt p ”
 p holds in the next state
 - $F p$: “Future p ”
 p holds somewhere on the subsequent path
 - $G p$: “Globally p ”
 p holds in all states of the subsequent path
 - $p U q$: “ p Until q ”
 p holds at least until q , which holds at the subsequent path

LTL temporal connectives

For a path of a Kripke structure



LTL examples I.

- $p \Rightarrow Fq$

If p holds in the initial state, then eventually q holds.

- Example: Start \Rightarrow F End

- $G(p \Rightarrow Fq)$

For all states, if p holds, then eventually q holds.

- Example: $G(\text{Request} \Rightarrow F \text{Reply})$
For a request, a reply always arrives

- $p U (q \vee r)$

Starting from the initial state, p holds until q or r eventually holds.

- Example: Requested U (Accept \vee Refuse)
A continuous request either gets accepted or refused

- $(p \wedge G(p \Rightarrow Xp)) \Rightarrow Gp$

Formalization of the mathematical induction principle - always holds

LTL examples II.

- GF p

After any states along the path, p will eventually hold

- There is no state after which p does not hold eventually
- Example: GF Running
The start state is reached from all states

- FG p

After some state, p will continuously hold.

- Example: FG Normal
After an initial transient the system operates normally

Formalizing requirements: Example

Consider an air conditioner with the following modes:

$AP = \{\text{Off, On, Error, MildCooling, StrongCooling, Heating, Ventilating}\}$

- Potentially more than one labels!
 - E.g. {On, Ventilating}
- When formalizing requirements, we might not yet know all potential behaviors
 - We assume only the labels on states

Example (cont.)

$AP = \{\text{Off, On, Error, MildCooling, StrongCooling, Heating, Ventilating}\}$

- The air conditioner can (and will) be turned on:
F On
- At some point, the air conditioner always breaks down
GF Error
- If the air conditioner breaks down, it eventually gets repaired
G (Error \Rightarrow F \neg Error)
- If the air conditioner breaks down, it cannot heat:
G \neg (Error \wedge Heating)

Example (cont.)

$AP = \{\text{Off, On, Error, MildCooling, StrongCooling, Heating, Ventilating}\}$

- The air conditioner can only break down when turned on:

G (**X** Error \Rightarrow On)

- After heating, the air conditioner must ventilate:

G ((Heating \wedge **X** \neg Heating) \Rightarrow **X** Ventilate)

but it may also break down:

G ((Heating \wedge **X** \neg Heating) \Rightarrow **X** (Ventilate \vee Error))

- After ventilation the air conditioner must not cool strongly until it performs some mild cooling:

G ((Ventilating \wedge **X** \neg Ventilating) \Rightarrow
X(\neg StrongCooling **U** MildCooling))

LTL formal interpretation

- So far we discussed the logic only informally
Questions arise, e.g.:
 - Does $F p$ hold if p holds in the first state?
 - Does $p U q$ hold if q holds in the first state?
- To enable formal verification, we need the following:
 - **Syntax:**
What are the well-formed formulas?
 - **Semantics:**
When does a given formula hold for a given model?

LTL syntax

The set of well-formed formulas (wff) in LTL are given as follows.

Let $P \in AP$ and p and q be wffs. Then

- **L1:** P is a wff.
- **L2:** $p \wedge q$ and $\neg p$ are wffs.
- **L3:** $p \cup q$ and Xq are wffs.

Precedence rules:

$X, U > \neg > \wedge > \vee > \Rightarrow > \equiv$

Derived connectives

- true holds for all states
false holds in no state
- $p \vee q$ means $\neg(\neg p \wedge \neg q)$
 $p \Rightarrow q$ means $\neg p \vee q$
 $p \equiv q$ means $p \Rightarrow q \wedge q \Rightarrow p$

- $F p$ means $\text{true} \cup p$
 $G p$ means $\neg F(\neg p)$
 $p \text{ WU } q$ means $G(p) \vee (p \cup q)$

- “Before” connective:

$$p \text{ WB } q = \neg((\neg p) \cup q)$$

$$p \text{ B } q = \neg((\neg p) \cup q) \wedge F q$$

Informally:

p must occur before q
(B : and q must occur)

(weak before)

(strong before)

LTL semantics – Notation

- $M = (S, I, R, L)$ Kripke structure
- $\pi = (s_0, s_1, s_2, \dots)$ a path of M where $s_0 \in I$ and $\forall i \geq 0: (s_i, s_{i+1}) \in R$
 - $\pi^i = (s_i, s_{i+1}, s_{i+2}, \dots)$ the suffix of π from i
- $M, \pi \models p$ denotes (logical entailment):
In Kripke structure M , along path π , p holds

The semantics of LTL defines when a wff holds over a path (i.e. it defines the entailment relation).

LTL semantics

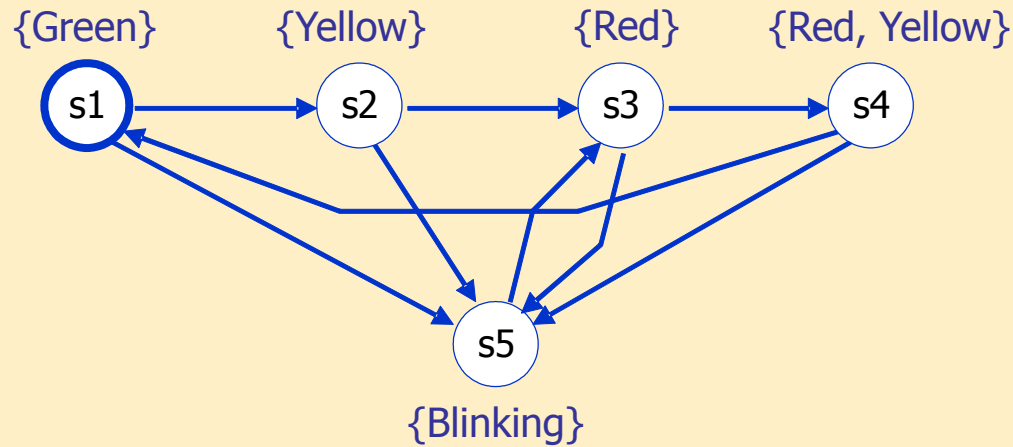
Defined recursively w.r.t. syntactic construction rules

- **L1:** $M, \pi \models P$ iff $P \in L(s_0)$
- **L2:** $M, \pi \models p \wedge q$ iff $M, \pi \models p$ and $M, \pi \models q$
 $M, \pi \models \neg q$ iff not $M, \pi \models q$.
- **L3:** $M, \pi \models (p \cup q)$ iff
 $\pi^j \models q$ for some $j \geq 0$ and
 $\pi^k \models p$ for all $0 \leq k < j$

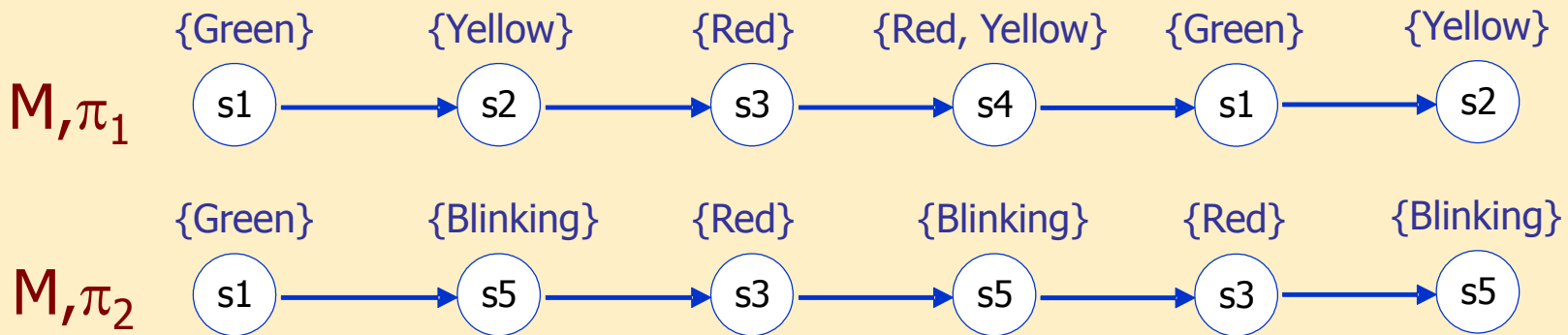
$$M, \pi \models X p \text{ iff } \pi^1 \models p$$

Interpreting LTL formulas, example

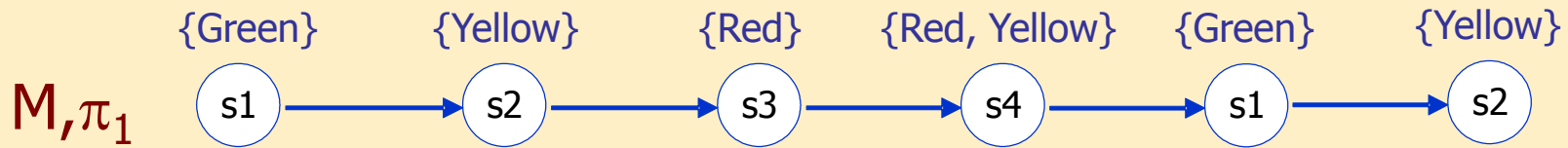
- Kripke structure M :



- Paths:

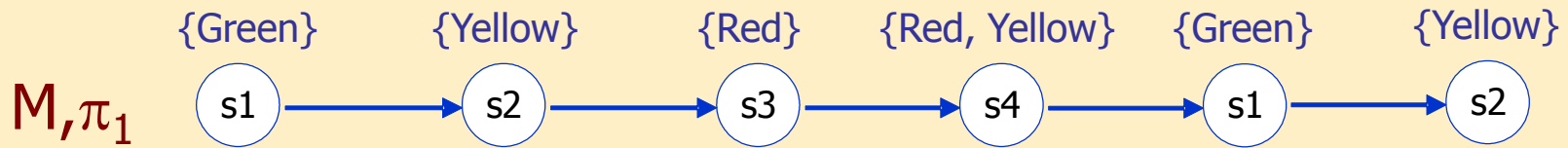


Examples (cont.)

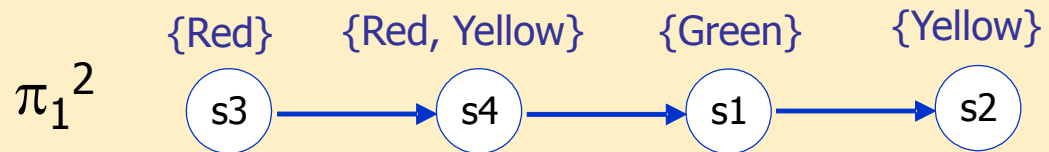


- $M, \pi_1 \models \text{Green}$, as $\text{Green} \in L(s_1)$
- not $M, \pi_1 \models \text{Red}$, as $\text{Red} \notin L(s_1)$
- not $M, \pi_1 \models \text{Green} \cup \text{Red}$,
as $\text{Red} \notin L(s_1)$, $\text{Red} \notin L(s_2)$ and $\text{Green} \notin L(s_2)$
- $M, \pi_1 \models \text{F Red}$, as $\text{Red} \in L(s_3)$
More precisely: $\pi_1^2 \models \text{Red}$

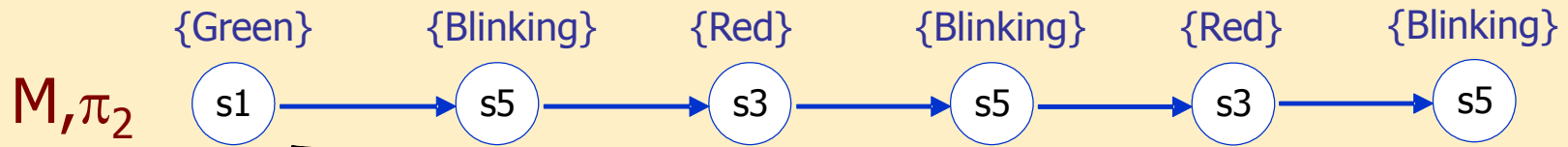
Examples (cont.)



- $M, \pi_1 \models F(\text{Red} \cup \text{Green})$,
as there exists a suffix for which
 $(\text{Red} \cup \text{Green})$ holds:

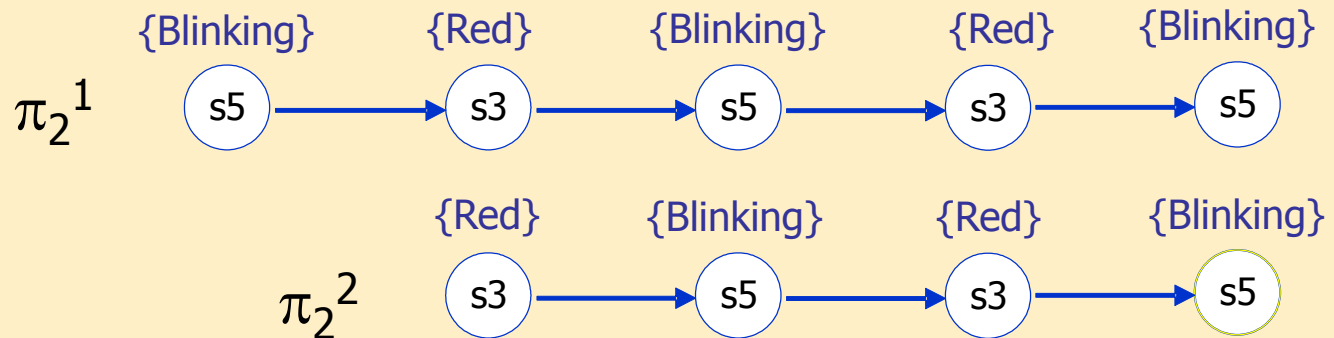


Examples (cont.)

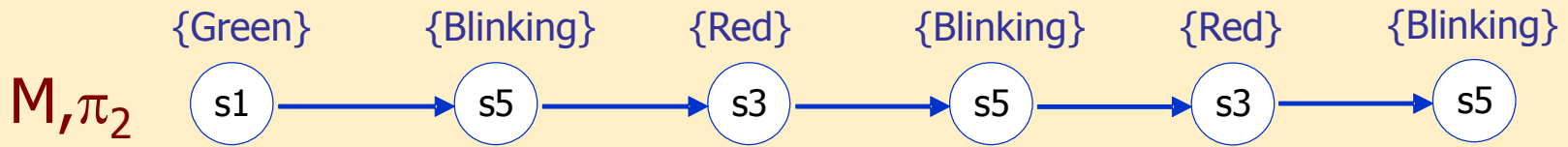


Also true for π_2^0 :
 $p \Rightarrow q$ is true if p is false

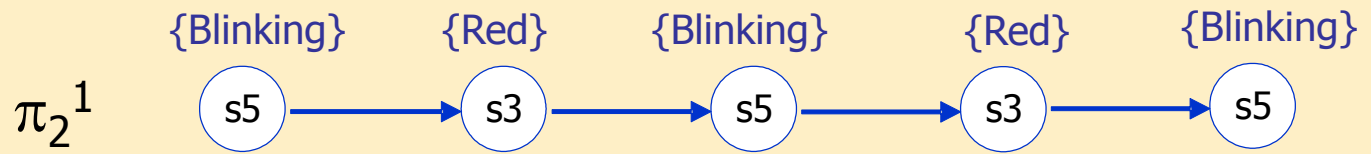
- $M, \pi_2 \models F (\text{Blinking} \Rightarrow X \text{Red})$,
 as there exists a suffix such that $\text{Blinking} \Rightarrow X \text{Red}$



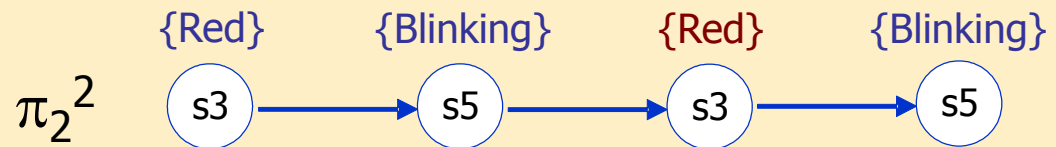
Examples (cont.)



- $M, \pi_2 \models XF (XX \text{ Red})$, as for suffix



$F (XX \text{ Red})$ holds, as
it has a suffix such that $XX \text{ Red}$ holds:



Extending LTL for LTSs

- Expresses properties of **transitions**:
labeling by **actions**
- Exactly one action per transition
- Application: modeling of communication and protocols

A labeled transition system *LTS* over a set of actions $Act = \{a, b, c, \dots\}$ is a triple (S, I, \rightarrow) where

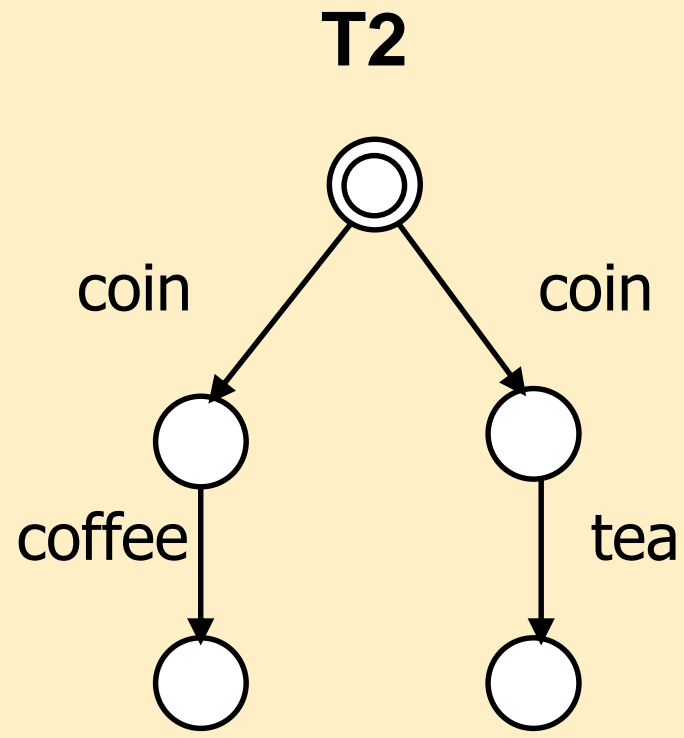
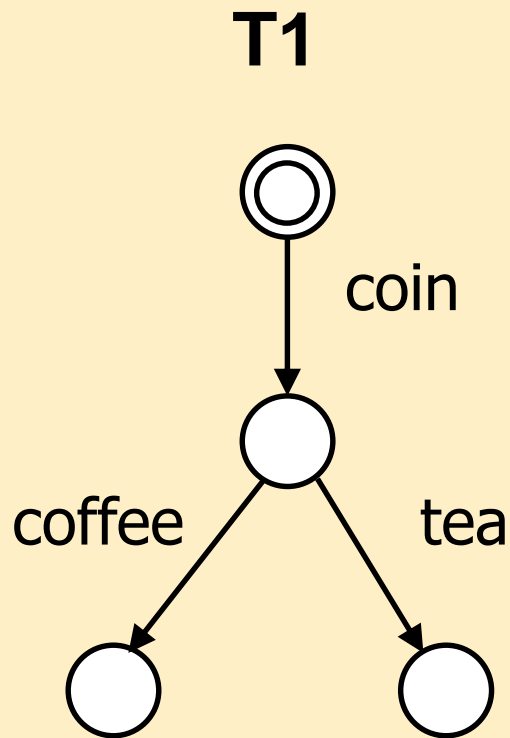
- $S = \{s_1, s_2, \dots, s_n\}$ is a finite set of states,
- $I \subseteq S$ is the set of initial states,
- $\rightarrow : S \times Act \times S$ is the set of transitions

We denote by $s \xrightarrow{a} s'$ iff $(s, a, s') \in \rightarrow$.

Example for LTS

Vending machine

- $Act = \{coin, coffee, tea\}$



LTL for LTSs

A path now is an alternating sequence of states and actions:

- $\pi = (s_0, a_1, s_1, a_2, s_2, a_3, \dots)$

Extending syntax:

- **L1***: If $a \in \text{Act}$ then (a) is a wff.

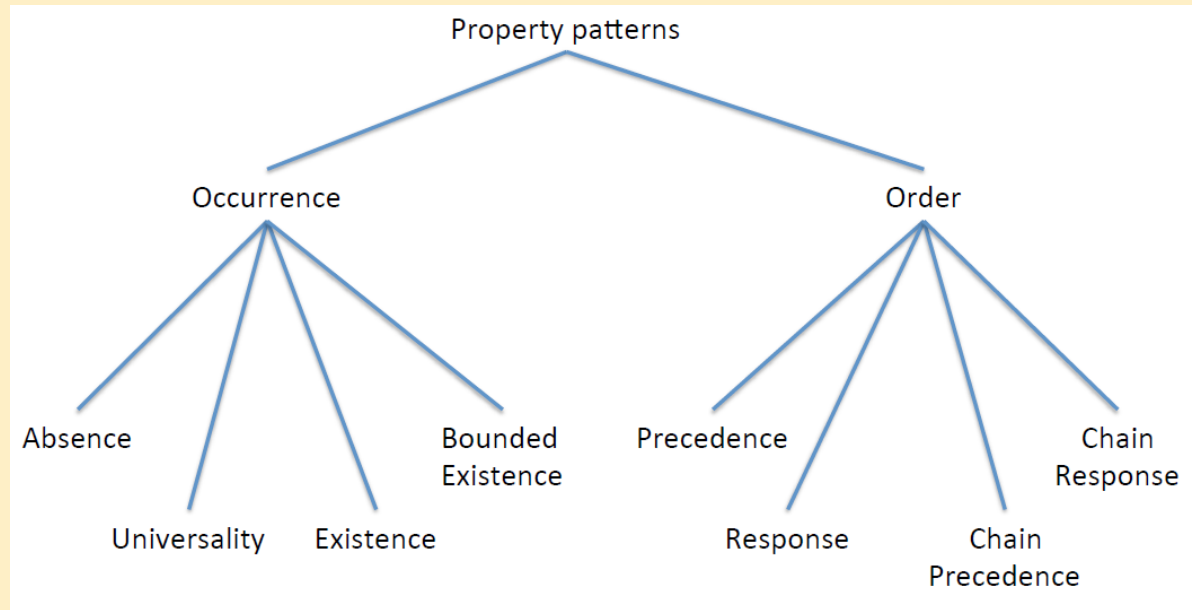
The corresponding case in semantics:

- **L1***: $M, \pi \models (a)$ iff. $a_1 = a$
where a_1 is the first action in π .

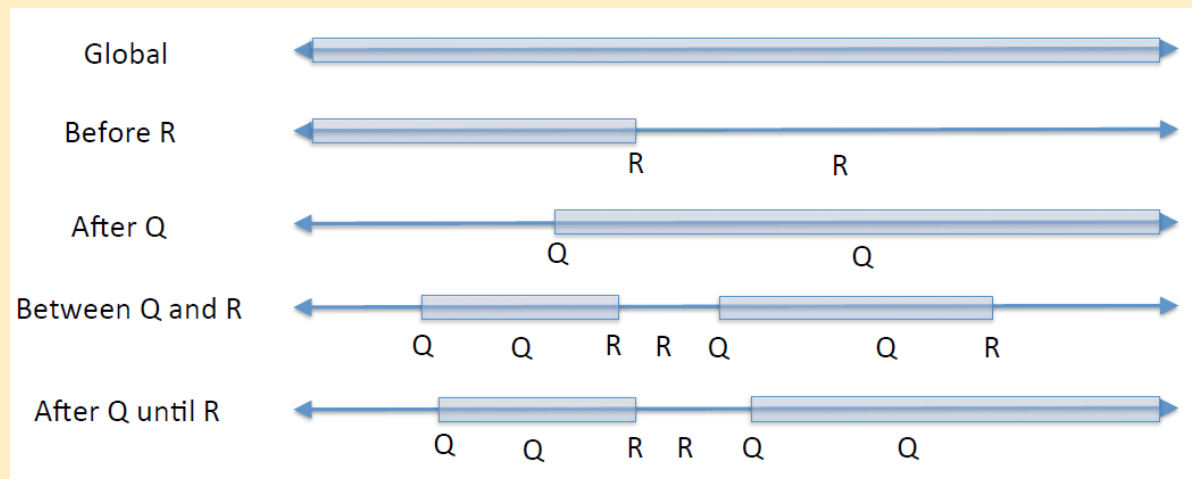
This way we can describe requirements of communicating systems.

Where we started: typical patterns for requirements

Pattern:
order or
occurrence



Scope:
relative to
further events



Formalization of patterns (examples)

After is inclusive,
Before is exclusive

Universality within scope	Property in LTL
P occurs in each step of the execution globally .	$G P$
P occurs in each step of the execution before Q .	$F Q \rightarrow (P U Q)$
P occurs in each step of the execution after Q .	$G(Q \rightarrow G P)$
P occurs in each step of the execution between Q and R .	$G((Q \wedge \neg R \wedge F R) \rightarrow (P U R))$

Existence within scope	Property in LTL
P occurs in the execution globally .	$F P$
P occurs in the execution before Q .	$\neg Q WU (P \wedge \neg Q)$
P occurs in the execution after Q .	$G (\neg Q) \vee F (Q \wedge F P)$
P occurs in the execution between Q and R .	$G((Q \wedge \neg R \wedge F R) \rightarrow (\neg R U (P \wedge \neg R)))$

Formalization of textual requirements (examples)

If α and β holds, then α has to remain true as long as β is true as well.

$$\mathbf{G}((\alpha \wedge \beta) \rightarrow (\alpha \mathbf{U} \neg\beta))$$

If a1arm is on and a1ert occurs, the output of safety should be true as long as a1arm is on.

$$\mathbf{G}((\text{alarm} = \text{ON} \wedge \text{alert}) \rightarrow \mathbf{X}(\text{safety} \mathbf{U} \neg\text{alarm}))$$

LTL summary

- Formalization of requirements
- Temporal logics
 - Linear temporal logic
 - Branching time temporal logic
- LTL
 - Connectives
 - Syntax
 - Semantics
- Interpretation of LTL formulas
- Formalization of requirements in LTL