Verification

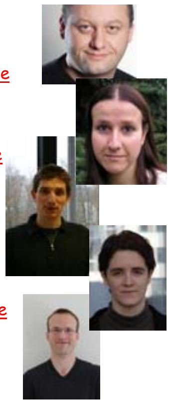
Bernd Finkbeiner, Sven Schewe,
Rayna Dimitrova, Lars Kuhtz,
Anne Proetzsch

Winter Semester 2007/2008

You've come to the right place.

- Core Lecture (Stammvorlesung)
- 9 LP
- Lecture:
  Tuesdays and Thursdays 14:15-15:45
  Building E 1 3, HS 002
- Tutorial A: Room 015 Bldg E 1 3, Wednesdays 14:00-16:00
- Tutorial B: Room 013 Bldg E 1 3, Fridays 10:00-12:00
  starting next week
- Web page
  react.cs.uni-sb.de/courses/verification

The Team.



- Bernd Finkbeiner
  E 1 3/506
  finkbeiner@cs.uni-sb.de
- Rayna Dimitrova
  E 1 3/507
  dimitrova@cs.uni-sb.de
- Lars Kuhtz
  E 1 3/532
  kuhtz@cs.uni-sb.de
- Anne Proetzsch
  E 1 3/532
  proetzsch@cs.uni-sb.de
- Sven Schewe
  E 1 3/508
  schewe@cs.uni-sb.de

This course is about…

Computer-Aided Verification

1. Verification:
   - find errors, or
   - prove that the system is correct.
2. Computer-Aided:
   - completely automatic, or
   - computer checks proof and helps with low-level details.

Context

Semantics → Verification
Concurrent programming → Verification
Logic → Verification
Automata theory → Verification

Verification

Verification → Theory: Specification logics, Model checking algorithms, proof systems
Verification → Tools: SPIN, STeP, SMV, PVS, UPPAAL, KRONOS, ACL2, Murφ, VIS, …
Verification → Applications: Protocols, distributed algorithms, medical equipment, power plants, …

Badmouth

- Verification can only be done by mathematicians.
- The verification process is itself prone to errors, so why bother?
- Using formal methods will slow down the project.

Some answers...

- Verification can only be done by mathematicians.

  Verification is based on mathematics
  but the user often does not need to know the math.

- The verification process is itself prone to errors, so why bother?

  Ultimately we reduce errors, we don't claim to eliminate them.

- Using formal methods will slow down the project.

  It may speed it up, if errors are found earlier.

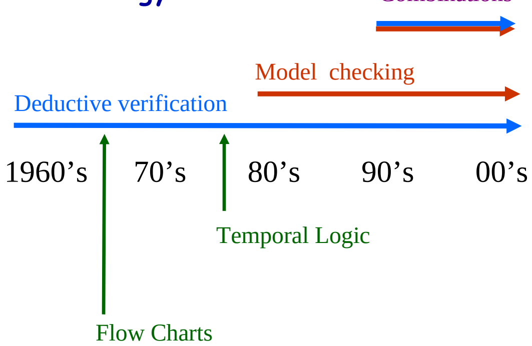
Verification Techniques

Deductive Verification:
Using some logical formalism, prove formally that the software satisfies its specification.

Model Checking:
Use some software to automatically check that the software satisfies its specification.

Testing:
Check computations of the software according to some coverage scheme. (Testing can only show the presence of errors, never the absence of errors. We won't focus on testing in this course.)

Chronology:



Deductive Verification	Combination	Model Checking
+ Infinite-state	+	Finite-state −
− Only proofs	+	Counterexamples +
− Interactive	+ −	Automatic +

(undecidable in general)

Popular exaggerations

- Model checking automatically finds errors.
- Deductive verification can show that the software is completely safe.

Course Outline

- Deductive Verification
  a proof system for linear-time temporal logic
- Model Checking
  linear and branching-time model checking
  automata over infinite objects
- Abstraction
  combining deductive verification and model checking
- Advanced Topics
  real-time/hybrid systems, game logics, ...

Course formalities…

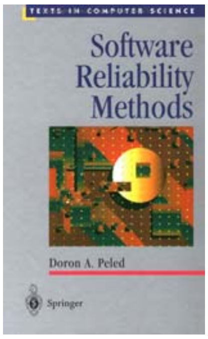
Homework Problems

- One problem set per week
- Mostly paper & pencil exercises, some exercises with tools
- Problem sets will be published on the web site on Tuesdays (first one *today*)
- Problem sets are due on Tuesday the following week *before the lecture*
- Will be discussed on Wednesday / Thursday
- We will try to return the problem sets as soon as possible
- Help us by submitting early ☺
- >50% points in Homework Problems -> Admission to final exam

Exam Policy

- **Midterm Exam: 20.12.2007**
- **Final Exam: 22.02.2008**
- **Backup Exam: 04.04.2008**
- **Requirement for admission to final exam:** more than 50% points in homeworks
- **Requirement for admission to backup exam:** passing grade in either midterm or final (but not both)
- *You pass the course if you pass two exams*
- *Your final grade is the average of the two passing grades*

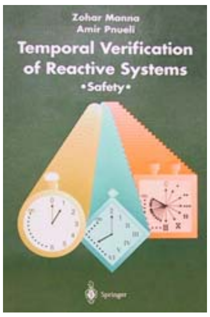
Literature Recommendations



Software Reliability Methods

by Doron A. Peled
Springer Verlag; ISBN: 0387951067

Literature Recommendations

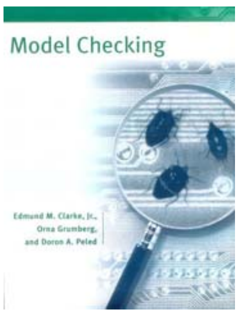


Temporal Verification of Reactive Systems – Safety

by Zohar Manna and Amir Pnueli

Springer Verlag; ISBN: 0387944591

Literature Recommendations



Model Checking

by Edmund M. Clarke, Jr.,
Orna Grumberg and Doron A. Peled

MIT Press; ISBN: 0262032708

Literature Recommendations

Principles of Model Checking

by Christel Baier and Joost-Pieter Katoen

To appear in Spring 2008

(we'll distribute selected chapters in class.)

Deductive Verification

Robert Floyd

Tony Hoare

Amir Pnueli

Zohar Manna

A first example

Peterson's Mutual Exclusion Algorithm

Version 1

Protection variables

local $\boxed{y_1, y_2}$: boolean where $y_1 = \mathrm{F}, y_2 = \mathrm{F}$

$P_1 ::$ ℓ_0: loop forever do
- ℓ_1: noncritical
- ℓ_2: $\boxed{y_1 := \mathrm{T}}$
- ℓ_3: $\boxed{\text{await } \neg y_2}$
- ℓ_4: critical
- ℓ_5: $\boxed{y_1 := \mathrm{F}}$

||

$P_2 ::$ m_0: loop forever do
- m_1: noncritical
- m_2: $y_2 := \mathrm{T}$
- m_3: await $\neg y_1$
- m_4: critical
- m_5: $y_2 := \mathrm{F}$

P_1 is interested

P_1 waits

P_1 resets y_1

Problem: deadlock possible

local y_1, y_2: boolean where $y_1 = \mathrm{F}, y_2 = \mathrm{F}$

$P_1 ::$ ℓ_0: loop forever do
- ℓ_1: noncritical
- ℓ_2: $y_1 := \mathrm{T}$
- ℓ_3: await $\neg y_2$
- ℓ_4: critical
- ℓ_5: $y_1 := \mathrm{F}$

||

$P_2 ::$ m_0: loop forever do
- m_1: noncritical
- m_2: $y_2 := \mathrm{T}$
- m_3: await $\neg y_1$
- m_4: critical
- m_5: $y_2 := \mathrm{F}$

1 2 3 4

May reach l_3, m_3 with

$y_1 = y_2 = \mathrm{T}$

Version 2

Signature variable

local y_1, y_2: boolean where $y_1 = \mathrm{F}, y_2 = \mathrm{F}$
s : integer where $s = 1$

$P_1 ::$ ℓ_0: loop forever do
- ℓ_1: noncritical
- ℓ_2: $(y_1, s) := (\mathrm{T}, 1)$
- ℓ_3: await $(\neg y_2) \vee \boxed{(s = 2)}$
- ℓ_4: critical
- ℓ_5: $y_1 := \mathrm{F}$

||

$P_2 ::$ m_0: loop forever do
- m_1: noncritical
- m_2: $(y_2, s) := (\mathrm{T}, 2)$
- m_3: await $(\neg y_1) \vee (s = 1)$
- m_4: critical
- m_5: $y_2 := \mathrm{F}$

P_1 requests priority

P_1 has priority
(P_2 was the last to request priority)

Properties of Version 2

- Mutual exclusion:

  P_1 and P_2 are never simultaneously in their critical sections

- 1-Bounded Overtaking:

  P_2 can visit its critical section at most once before P_1 gets to visit its critical section (if P_1 is waiting).

- Accessibility:

  If P_1 leaves the noncritical section it will eventually enter the critical section.

Version 2

local y_1, y_2: boolean where $y_1 = \text{F}, y_2 = \text{F}$
s : integer where $s = 1$

$P_1 ::$
ℓ_0: loop forever do
ℓ_1: noncritical
ℓ_2: $\boxed{(y_1, s) := (\text{T}, 1)}$
ℓ_3: await $(\neg y_2) \vee (s = 2)$
ℓ_4: critical
ℓ_5: $y_1 := \text{F}$

||

$P_2 ::$
m_0: loop forever do
m_1: noncritical
m_2: $(y_2, s) := (\text{T}, 2)$
m_3: await $(\neg y_1) \vee (s = 1)$
m_4: critical
m_5: $y_2 := \text{F}$

Simultaneous assignment may not be available.

Version 3: split assignment

Version 3

local y_1, y_2: boolean where $y_1 = \text{F}, y_2 = \text{F}$
s : integer where $s = 1$

$P_1 ::$
ℓ_0: loop forever do
ℓ_1: noncritical
ℓ_2: $s := 1$
ℓ_3: $y_1 := \text{T}$
ℓ_4: await $(\neg y_2) \vee (s = 2)$
ℓ_5: critical
ℓ_6: $y_1 := \text{F}$

||

$P_2 ::$
m_0: loop forever do
m_1: noncritical
m_2: $s := 2$
m_3: $y_2 := \text{T}$
m_4: await $(\neg y_1) \vee (s = 1)$
m_5: critical
m_6: $y_2 := \text{F}$

Problem: Violation of Mutual Exclusion

local y_1, y_2: boolean where $y_1 = \text{F}, y_2 = \text{F}$
s : integer where $s = 1$

$P_1 ::$
ℓ_0: loop forever do
ℓ_1: noncritical
1
ℓ_2: $s := 1$
2
ℓ_3: $y_1 := \text{T}$
7
ℓ_4: await $(\neg y_2) \vee (s = 2)$
8
ℓ_5: critical
ℓ_6: $y_1 := \text{F}$

||

$P_2 ::$
m_0: loop forever do
m_1: noncritical
3
m_2: $s := 2$
4
m_3: $y_2 := \text{T}$
5
m_4: await $(\neg y_1) \vee (s = 1)$
6
m_5: critical
m_6: $y_2 := \text{F}$

May reach l_5, m_5

Version 4

local y_1, y_2: boolean where $y_1 = \text{F}, y_2 = \text{F}$
s : integer where $s = 1$

$P_1 ::$
ℓ_0: loop forever do
ℓ_1: noncritical
ℓ_2: $y_1 := \text{T}$
ℓ_3: $s := 1$
ℓ_4: await $(\neg y_2) \vee (s = 2)$
ℓ_5: critical
ℓ_6: $y_1 := \text{F}$

||

$P_2 ::$
m_0: loop forever do
m_1: noncritical
m_2: $y_2 := \text{T}$
m_3: $s := 2$
m_4: await $(\neg y_1) \vee (s = 1)$
m_5: critical
m_6: $y_2 := \text{F}$

Swap comands

Correct?

Modeling Software Systems

Types of Systems

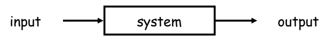
- Sequential systems.
- Reactive systems.

We'll focus on reactive systems

  - Distributed systems.
  - Concurrent systems.
  - Embedded systems (software + hardware).

Sequential Systems

- Observable only at the beginning and end of their execution

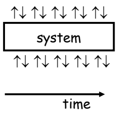


  with no interaction with the environment.

- Sequential systems typically terminate.
- Specified by input-output relations
  → First-Order Logic.

Reactive Systems

- Observable throughout their execution



- Reactive Systems typically don't terminate.
- Interaction with environment specified by the ongoing behavior (interaction history)
  → automata, temporal logic

States

- Vocabulary $\mathcal{V}$
  a set of typed variables
  - Expressions over $\mathcal{V}$ x+y
  - Assertions over $\mathcal{V}$ x>y
- A state is an interpretation over $\mathcal{V}$

  Example:
  $\mathcal{V}$ = {x,y}
  s = <x:1, y:2>
  (also written as s[x]=1, s[y]=2)
  x>y is false on s

- Set of all states: Σ

Transition Systems

- A (finite) set of variables $V \subseteq \mathcal{V}$
  System variables: data variables + control variables
- Initial condition θ
  first-order assertion over
  that characterizes all initial states
- A (finite) set of transitions $\mathcal{T}$

Transitions

For each $\tau \in \mathcal{T}$: $\tau : \Sigma \mapsto 2^{\Sigma}$

(each transition is a function from states to sets of states)

- s' is a τ-successor of s if s' $\in \tau(s)$
- τ is represented by the transition relation $\rho(\tau)$
  (next-state relation)

  V values of variables in the current state

  V' values of variables in the next state

Enabled/Disabled/Taken Transitions

- A transition τ
  - is enabled on s if $\tau(s) \neq \{\}$
  - is disabled on s if $\tau(s) = \{\}$
- For an infinite sequence of states

  σ: $s_0, s_1, s_2, \ldots$

  a transition τ
  - is enabled at position k if it is enabled on s_k
  - is taken at position k if s_{k+1} is a τ-successor of s_k

The Interleaving Model

Infinite sequence of states

σ: $s_0, s_1, s_2, \ldots$

is a run of a transition system, if it satisfies the following:

- Initiality: s_0 satisfies θ
- Consecution: For each i= 0,1, …

there is a transition $\tau \in \mathcal{T}$ s.t. $s_{i+1} \in \tau(s_i)$

Example

- V : {a, b, c, d, e: integer}
- θ : c=a $\wedge$ d=b $\wedge$ e=0
- $\mathcal{T}$: $\{\tau_1, \tau_2\}$
- $\rho(\tau_1)$: c>0 $\wedge$ c'=c-1 $\wedge$ e'= e+1
- $\rho(\tau_2)$: d>0 $\wedge$ d'=d-1 $\wedge$ e'= e+1

- s_0=<a=2, b=1, c=2, d=1, e=0>
  (satisfies the initial condition)
  (first transition taken)
- s_1=<a=2, b=1, c=1, d=1, e=1>
  (second transition taken)
- s_2=<a=2, b=1, c=1, d=0, e=2>
  (first transition taken again)
- s_3=<a=2, b=1 ,c=0, d=0, e=3>

Idling Transition

- What if no transition is enabled?
- We implicitly assume that there is an idling transition (stuttering transition) τ_I

  $\rho(\tau_I)$: V = V'
- The idling transition is always enabled.

Reachable States

For a transition system Φ,
a state s is Φ-accessible if there is a run

σ: $s_0, s_1, s_2, \ldots$

with s=s_i, for some i.

A transition system Φ is finite-state if the set of all Φ-accessible states is finite.

Atomic Transitions

- Each atomic transition represents a small piece of code such that no smaller piece of code is observable.
- Is a:=a+1 atomic?
- In some systems, e.g., when a is a register and the transition is executed using an inc command.

Non-atomicity

- Execute the following when a=0 in two concurrent processes:

```
P1:a=a+1
P2:a=a+1
```

- Result: a=2.
- Is this always the case?

- Consider the actual translation:

```
P1:load R1,a
   inc R1
   store R1,a
P2:load R2,a
   inc R2
   store R2,a
```

- a may also be 1.

The Scheduler

- Start from some initial state s_0 such that s_0 satisfies θ.
- Set s = s_0.
- Loop forever:
  - Pick a transition τ that is currently enabled at s.
  - Select a new state state s' in τ(s).
  - Set s=s'.

Nondeterministic choice.

Fairness?

Fair Transition Systems

$$\Phi = (V, \theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$$

- $\mathcal{J} \subseteq \mathcal{T}$: set of just (weakly fair) transitions
- $\mathcal{C} \subseteq \mathcal{T}$: set of compassionate (strongly fair) transitions

- Justice: for each just transition it is not the case that the transition is continually enabled but only taken at finitely many positions.
- Compassion: for each compassionate transition it is not the case that the transition is enabled at infinitely many positions but only taken at finitely many positions.

Example

- V : {x,y: integer}
- θ : x=0 $\wedge$ y=0
- $\mathcal{T}$: $\{\tau_I, \tau_x, \tau_y\}$
- $\mathcal{J}$: $\{\tau_x\}$
- $\mathcal{C}$: $\{\tau_y\}$
- $\rho(\tau_x)$: x' = x+1 mod 2
- $\rho(\tau_y)$: x=1 $\wedge$ y' = y+1

- s_0=<x=0, y=0> (satisfies the initial condition)
- s_1=<x=1, y=0> (τ_x taken)
- s_2=<x=0, y=0> (τ_x taken)
- s_3=<x=1, y=0> (τ_x taken)
- …

Justice: YES

Compassion: NO (τ_y is infinitely often enabled but never taken.)

Computations

An infinite sequence of states

$$\sigma: s_0, s_1, s_2, \ldots$$

is a computation of a fair transition system, if it satisfies:

- Initiality
- Consecution
- Justice
- Compassion

Fairness = Justice + Compassion
Computation = Run + Fairness

Specifying Properties in Linear Time Temporal Logic

Temporal Logic

Two views:

- Linear Time Temporal Logic: LTL
  - Program generates infinite sequences of states
  - Models of LTL formulas are infinite sequences of states
- Computation Tree Logic
  - Program generates an infinite tree, where branching points represent nondeterminism in the program
  - Models of CTL formulas are infinite trees.

We'll continue with LTL and return to CTL later in the course.

LTL

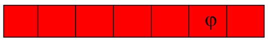
- LTL is defined relative to an underlying assertion language, in which conditions over individual states are formulated.
- For example: propositional logic, first-order logic.
- In this part of the course, we use a first-order language over interpreted symbols (functions and relations over concrete domains).

  Example: x>5

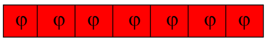
  Formulas of this language are called state formulas or assertions.

Temporal Operators

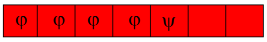
$\Diamond\varphi$ Eventually



$\Box\varphi$ Henceforth

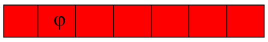


$\varphi\,\mathcal{U}\,\psi$ Until



$\varphi\,\mathcal{W}\,\psi$ Wait-for $\Box\varphi \vee \varphi\,\mathcal{U}\,\psi$

$\bigcirc\varphi$ Next



LTL Syntax

- Every assertion is a temporal formula.
- If φ and ψ are temporal formulas, then so are

$$\neg\varphi \qquad \varphi\vee\psi \qquad \varphi\wedge\psi$$

$$\Diamond\varphi \qquad \Box\varphi \qquad \varphi\,\mathcal{U}\,\psi \qquad \varphi\,\mathcal{W}\,\psi \qquad \bigcirc\varphi$$

LTL Semantics

- LTL formulas are evaluated over an infinite sequence of states

  σ: $s_0, s_1, s_2, \ldots$

- The semantics of an LTL formula φ is defined inductively at position $j \geq 0$

$$(\sigma, j) \vDash \varphi$$

LTL Semantics

p evaluated locally using the interpretation of s_j

- For state formulas:

$$(\sigma, j) \vDash p \iff s_j \vDash p$$

- For temporal formulas:

$$(\sigma, j) \vDash \neg p \iff (\sigma, j) \nvDash p$$

$$(\sigma, j) \vDash p \vee q \iff (\sigma, j) \vDash p \text{ or } (\sigma, j) \vDash q$$

LTL Semantics

- $(\sigma, j) \vDash \Box p \iff$
  for all $k \geq j,\ (\sigma, k) \vDash p$
- $(\sigma, j) \vDash \Diamond p \iff$
  for some $k \geq j,\ (\sigma, k) \vDash p$
- $(\sigma, j) \vDash p \mathcal{U} q \iff$
  for some $k \geq j,\ (\sigma, k) \vDash q$,
  and for all $i,\ j \leq i < k,\ (\sigma, i) \vDash p$
- $(\sigma, j) \vDash p \mathcal{W} q \iff$
  $(\sigma, j) \vDash p \mathcal{U} q$ or $(\sigma, j) \vDash \Box p$
- $(\sigma, j) \vDash \bigcirc p \iff$
  $(\sigma, j+1) \vDash p$

Examples

- $p \rightarrow \Diamond q$

  if initially p then eventually q
- $\Box(p \rightarrow \Diamond q)$

  every p is eventually followed by a q
- $\Box \Diamond q$

  infinitely many q

Examples

- $\Diamond \Box q$

  finitely many $\neg$q
- $\Box \Diamond p \rightarrow \Box \Diamond q$

  if there are infinitely many p
  then there are infinitely many q
- $(\neg p) \mathcal{W} q$

  q precedes p

Abbreviations

- $p \Rightarrow q$ stands for $\Box(p \rightarrow q)$

  (entailment)
- $p \approx q$ stands for $\Box(p \leftrightarrow q)$

  (congruence)
- $q_1 \mathcal{W} q_2 \mathcal{W} q_3 \mathcal{W} q_4$ stands for $q_1 \mathcal{W} (q_2 \mathcal{W} (q_3 \mathcal{W} q_4))$

  (nested waiting-for)

Peterson's Algorithm

local y_1, y_2: boolean where $y_1 = \text{F}, y_2 = \text{F}$
s : integer where $s = 1$

$P_1 ::$
ℓ_0: loop forever do
$$\begin{bmatrix} \ell_1: & \textbf{noncritical} \\ \ell_2: & (y_1, s) := (\text{T}, 1) \\ \ell_3: & \textbf{await } (\neg y_2) \vee (s = 2) \\ \ell_4: & \textbf{critical} \\ \ell_5: & y_1 := \text{F} \end{bmatrix}$$

||

$P_2 ::$
m_0: loop forever do
$$\begin{bmatrix} m_1: & \textbf{noncritical} \\ m_2: & (y_2, s) := (\text{T}, 2) \\ m_3: & \textbf{await } (\neg y_1) \vee (s = 1) \\ m_4: & \textbf{critical} \\ m_5: & y_2 := \text{F} \end{bmatrix}$$

- Mutual exclusion:

  $\Box \neg(\text{at\_}\ell_4 \wedge \text{at\_}m_4)$
- 1-Bounded Overtaking:

  $\text{at\_}\ell_3 \Rightarrow (\neg\text{at\_}m_4) \mathcal{W} \text{at\_}m_4 \mathcal{W} (\neg\text{at\_}m_4) \mathcal{W} \text{at\_}\ell_4$
- Accessibility:

  $\text{at\_}\ell_3 \Rightarrow \Diamond \text{at\_}\ell_4$
- Communal Accessibility:

  $\text{at\_}\ell_3 \vee \text{at\_}m_3 \Rightarrow \Diamond \text{at\_}\ell_4 \vee \text{at\_}m_4$

Satisfiability / Validity

- For a temporal formula p
  and sequence σ,

  $\sigma \vDash p$ iff $(\sigma, 0) \vDash p$
- The formula p is satisfiable if $\sigma \vDash p$ for some sequence σ
- The formula p is valid if $\sigma \vDash p$ for all sequences σ

Examples

- $\Box x<5$

  is satisfiable

- $\Box x<5 \;\vee\; \Diamond x\geq 5$

  is valid

- $\Box x<5 \;\wedge\; \Diamond x\geq 5$

  is unsatisfiable

Congruences

$$
\begin{array}{rcl}
\Box(p\wedge q) & \approx & \Box p \wedge \Box q \\
\Diamond(p\vee q) & \approx & \Diamond p \vee \Diamond q \\
p\,\mathcal{U}\,(q\vee r) & \approx & p\,\mathcal{U}\,q \vee p\,\mathcal{U}\,r \\
(p\wedge q)\,\mathcal{U}\,r & \approx & p\,\mathcal{U}\,r \wedge q\,\mathcal{U}\,r \\
p\,\mathcal{W}\,(q\vee r) & \approx & p\,\mathcal{W}\,q \vee p\,\mathcal{W}\,r \\
(p\wedge q)\,\mathcal{W}\,r & \approx & p\,\mathcal{W}\,r \wedge q\,\mathcal{W}\,r \\
\Box p & \approx & (p \wedge \bigcirc\Box p) \\
\Diamond p & \approx & (p \vee \bigcirc\Diamond p) \\
p\,\mathcal{U}\,q & \approx & [q \vee (p \wedge \bigcirc(p\,\mathcal{U}\,q))]
\end{array}
$$

Expressiveness

There are properties (i.e., sets of sequences) that cannot be expressed as LTL formulas.

<u>Example:</u> „x=0 is true only at even positions" cannot be expressed.

Note: „x=0 is true exactly at the even positions" can be expressed!

$$x=0 \wedge \Box\,((x=0) \leftrightarrow \bigcirc(x\neq 0))$$

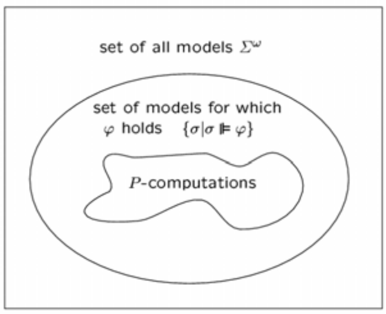
LTL & Programs

P-Validity

- A LTL formula φ is valid over a program P, written $P \models \varphi$,

  if φ holds in the first state of every computation of P.

P-Validity



P-Validity

	general	program P
state formula q	$\Vdash q$ state valid „q holds in all states"	$P \Vdash q$ P-state valid „q holds in all P-accessible states"
temporal formula φ.	$\vDash \varphi$ Valid „φ holds in the first position of every sequence"	$P \vDash \varphi$ P-valid „φ holds in the first position of every P-computation"