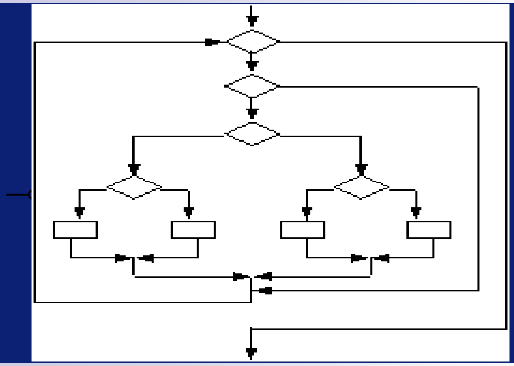
Spring 2008



Program correctness

Overview and motivation

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

Course Information

- My e-mail: marcello@liacs.nl
- My office: 155a

- All important information on www.liacs.nl/~marcello/penc.html
  - Schedule
  - Grades
  - …

Visit it regularly

Lectures

- Where: room WI312
- When: Friday (11:15 - 13:00)

February	8	15	22	29
March	7	14		28
April	~~4~~	11	18	25
May		9	16	

Class participation is important

Practice

- Where: room WI312
- When: Friday (13:45 - 15:30)

February	8	15	22	29
March	7	14		28
April	4	11	18	25
May		9	16	

Class participation is essential

Grading

- This course is worth 5 ECTS

- Evaluation by home assignment (20%)
  +
  written examination (80%)

  max {0.2*HA + 0.8 WE, 1.0*WE}

- Written examination
  - when: Wednesday 11 June from 14:00 to 17:00
  - where: room ???

  and also

  - when: Thursday 21 Augustus from 10:00 to 13:00
  - where: room ???

Reading

Logic in Computer Science:

Modelling and Reasoning about Systems

Michael R. A. Huth and Mark D. Ryan

Cambridge University Press, 2004

ISBN 0 521 54310 X paperback

Expected Background

- Propositional logic
- Predicate logic
- Sets and functions
- Induction
- Recursion

- Imperative programming

Course Organization

- The course cover model and proof-based techniques for proving programs correct

- The course combines
  - theory (logics)
  - practice (program and system modeling)

- Course goals:
  - introduction to fundamental concepts of formal methods
  - usage of formal methods in software engineering

Formal Methods

- Formal methods includes all applications of mathematics to software engineering problems.

  - type checking
  - model checking

  - program correctness
  - semantics

Formal Methods

- We consider formal methods for verifying the correctness of computer systems (hardware and/or software)
- Logics provide a mean for mechanizing verification details

computer aided verification

  - ☐ fully automated (e.g. model checking)
  - ☐ interactive (e.g. program correctness)

Why?

- Avoid loss of life

Therac 25, a computer-controlled radiation therapy machine made by Atomic Energy of Canada killed 6 people by radiation overdoses between 1985 and 1987 because of a timing problem on a data entry:

“An operator mistake could be fixed within 8 seconds, but even though the monitor reflected the operator change, the change did not affected a part of the program”

Why?

- Save costs

  In 1994, 2 million Intel Pentium V had a bug in the FDIV operation. It could be detected by the following MS-Excel operation:

(4195835/3145727)x3145727-4195835 = 512 !!!

  - Cost to Intel: $475 million
  - From 1994 Intel applies formal verification techniques to its products

Why?

- Guarantee security

In 1998 several e-mail systems did not check for the length of e-mail addresses, and allowed their buffers to overflow causing the applications to crash.

Hostile hackers used this fault to trick the operating system into running a malicious program in its place.

Do you trust your system?

The real wonder is that the system works

as well as it does

(Peterson, 1996)

but remember that software systems provide the infrastructure in virtually all industries today:

- air traffic control
- water level management
- energy production and distribution
- ...

Why Formal Methods?

- Testing and simulation techniques are never exhaustive

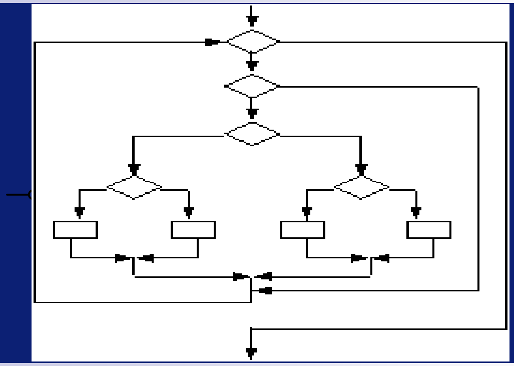
- Formal verification proves that a system works based on:
  - mathematical principles
  - exhaustive verification techniques
  - mathematical model structures

Warning

- The use of formal methods does not solve all these problems
  - ☐ proof: hand-checked or machine supported?
  - ☐ modelling task: difficult and yet crucial!

- Formal methods should be part of a methodology together with
  - ☐ Reviews (of requirements, design, and code)
  - ☐ Testing (of software units and their integration)

Spring 2007



Program correctness

Transition systems

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

Formal Verification

Verification techniques comprise

- a modelling framework M, Γ

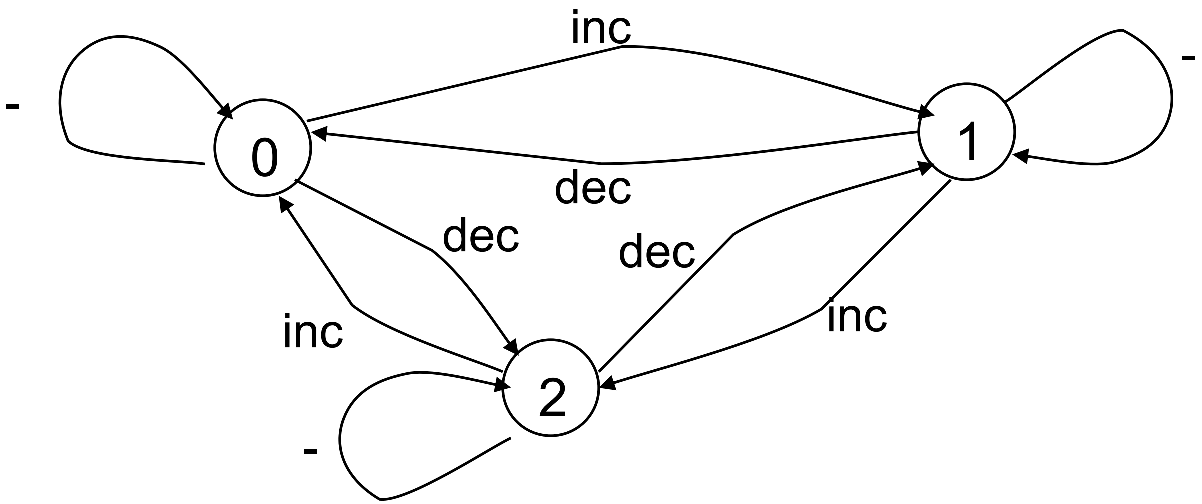
  to describe a system
- a specification language ϕ

  to describe the properties to be verified
- a verification method $M \models \phi, \Gamma \vdash \phi$

  to establish whether a model satisfies a property

Transition Systems

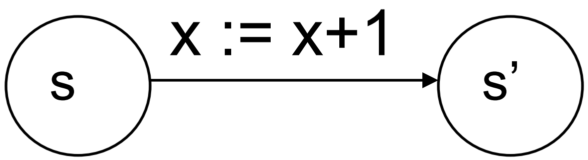
- A very general modelling framework
- Intuitively: a system evolves from one state to another under the action of a transition



A modulo 3 counter

Example: an assignment

States: s:Var -> Val



x := x+1

s

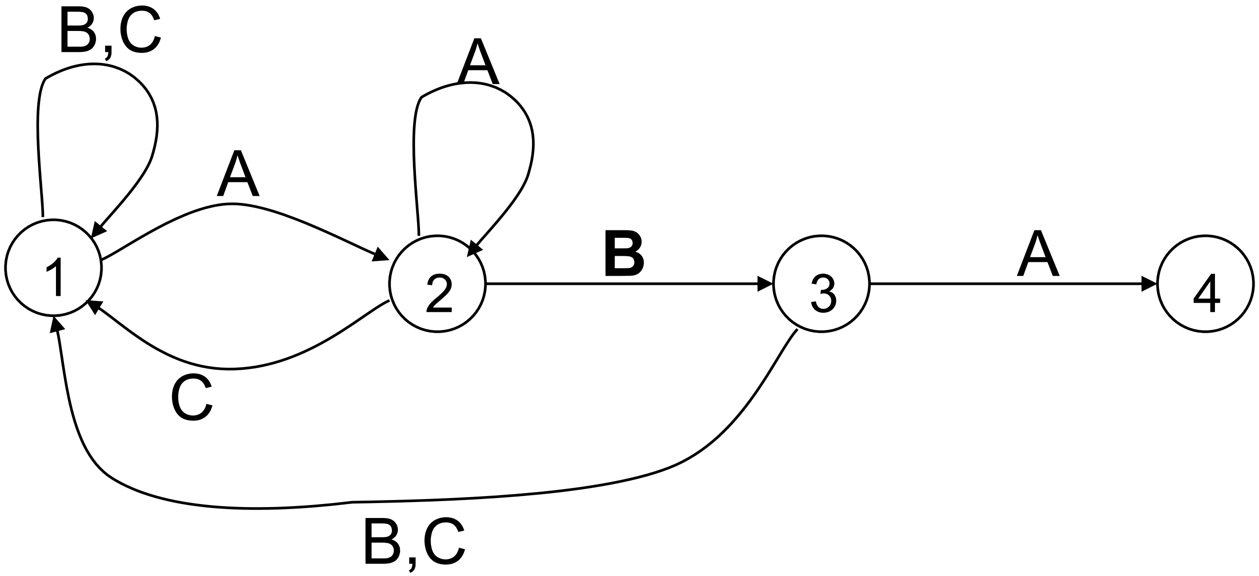
s’

where s’=s[s(x)+1/x] and

$$f[v/x](y) = \begin{cases} f(y) & \text{if } x \neq y \\ v & \text{if } x = y \end{cases}$$

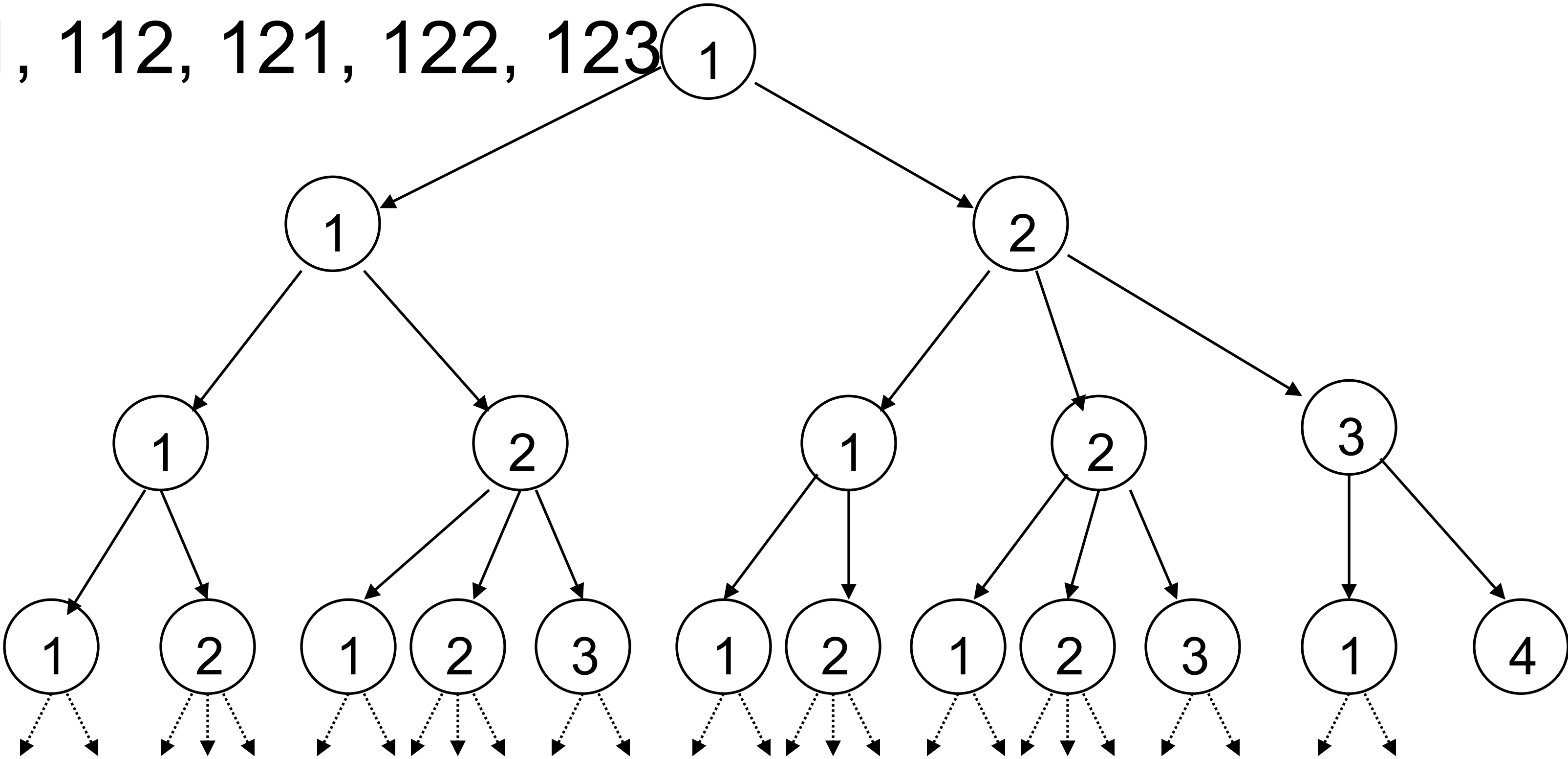
Example: a digicode

- 3 keys: A, B, C
- The door open when ABA is keyed



Digicode's executions

- 1
- 11, 12
- 111, 112, 121, 122, 123
- …



A few definitions

- Transition system: $<S,L,\rightarrow>$

S	set of states
L	set of transition labels
$\rightarrow \subseteq S \times L \times S$	transition relation

- Path: a sequence π of infinite transitions which follow each other

For example

$$3 \xrightarrow{B} 1 \xrightarrow{A} 2 \xrightarrow{A} 2 \ldots$$

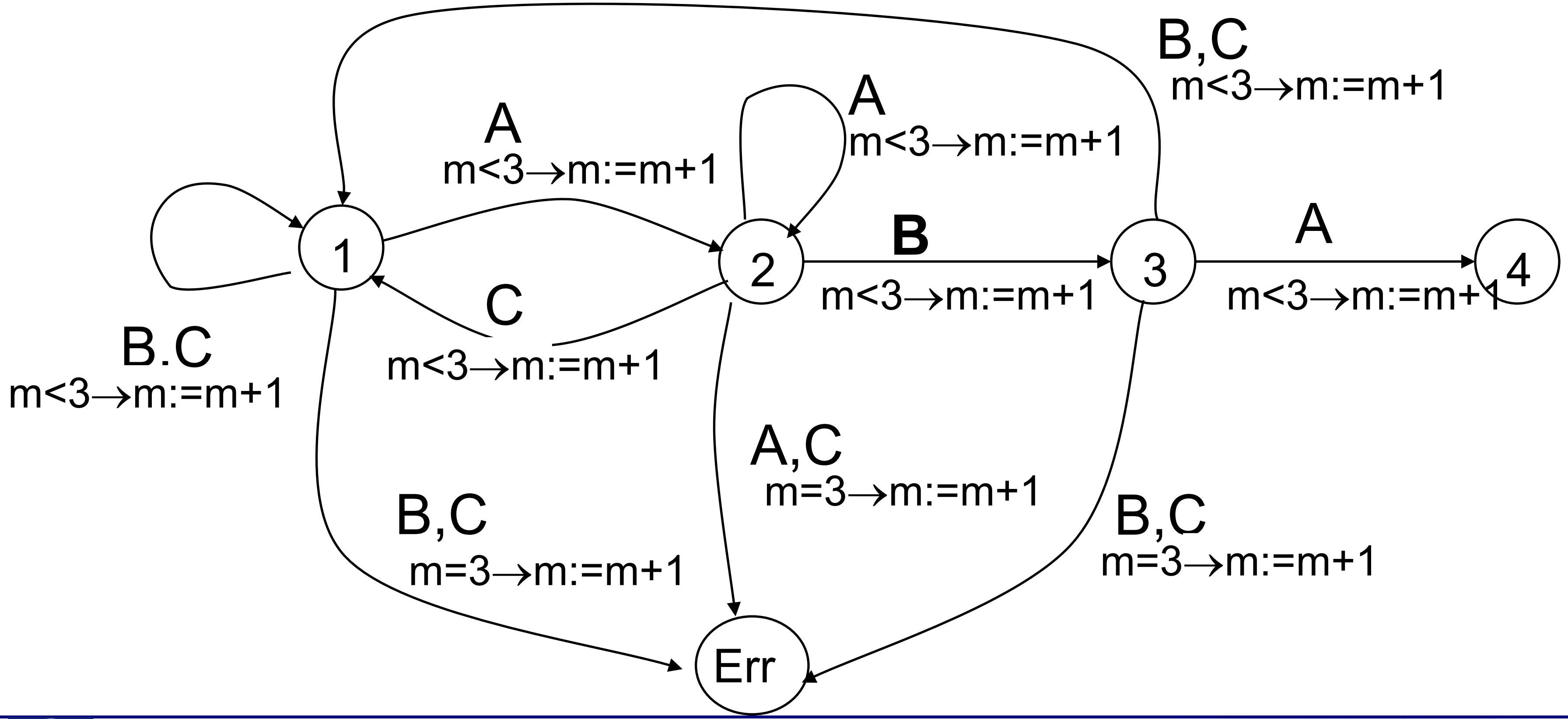
is a path of the digicode

Adding data

- Real-life systems consist of control and data. We can model them by
  - control = states+transitions
  - data = state variables
- A transition system interact with state variables in two ways
  - guards a transition cannot occur if the condition does not holds
  - assignment a transition can modify the value of some state variables

Back to the digicode

- We do not tolerate more than 3 mistakes (recorded by the variable m)



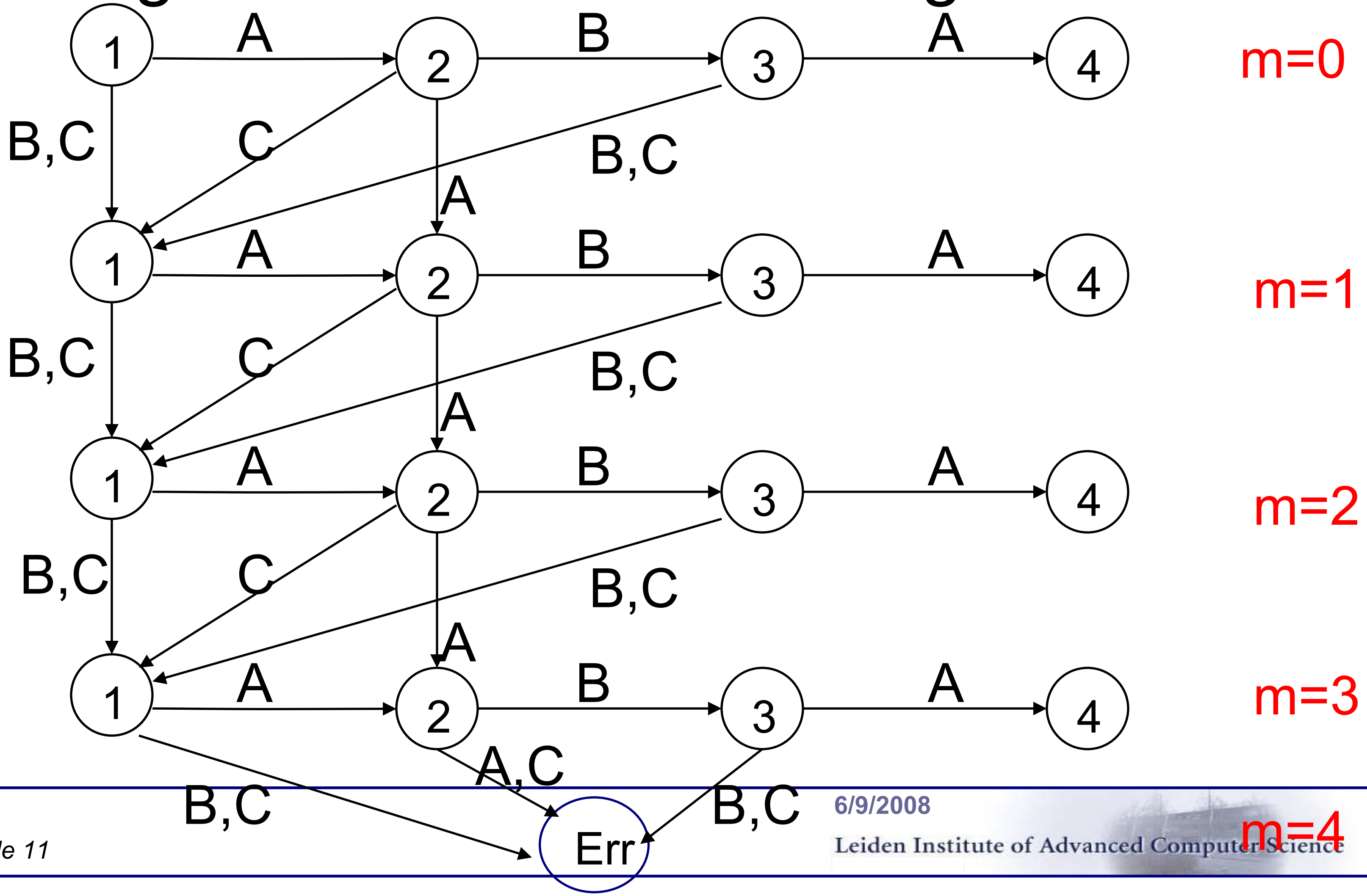
Unfolding

From a theoretical point of view, transition systems with state variables are not strictly necessary, as we can unfold them into ordinary transition systems.

- The new states correspond to the old ones + a component for each variable giving its value
- no more guards and assignment on the new transitions

Unfolding: example

- The digicode with error counting



Composing systems

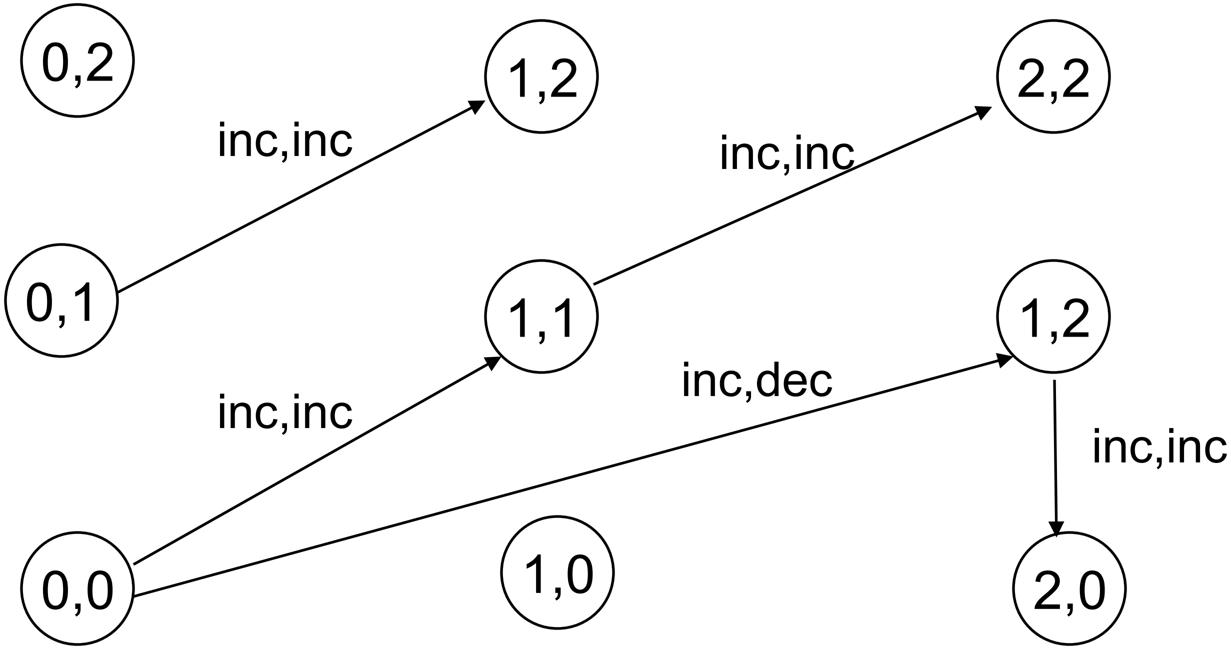
- Systems often consists of cooperating subsystems. Next we describe how to obtain a global transition system form its subsystem by having them cooperate
- There are many ways to cooperate:
  - product (no interaction)
  - synchronous product
    - by message passing
    - by asynchronous channels
    - by shared variables

Product

- Subsystems do not interact with each other
- The resulting transition system <S,L,→> is the cartesian product of the transition systems <S1,L1,→> ,…, <Sn,Ln,→> representing the subsystems
  - $S = S1 \times \ldots \times Sn$
  - $L = L1 \times \ldots \times Ln$
  - $<s_1,\ldots,s_n> \xrightarrow{<e_1,\ldots,e_n>} <t_1,\ldots t_n>$ if for all i, $s_i \xrightarrow{e_i} t_i$

Example

- Few transitions of the product of two modulo 3 counters



Synchronized Product

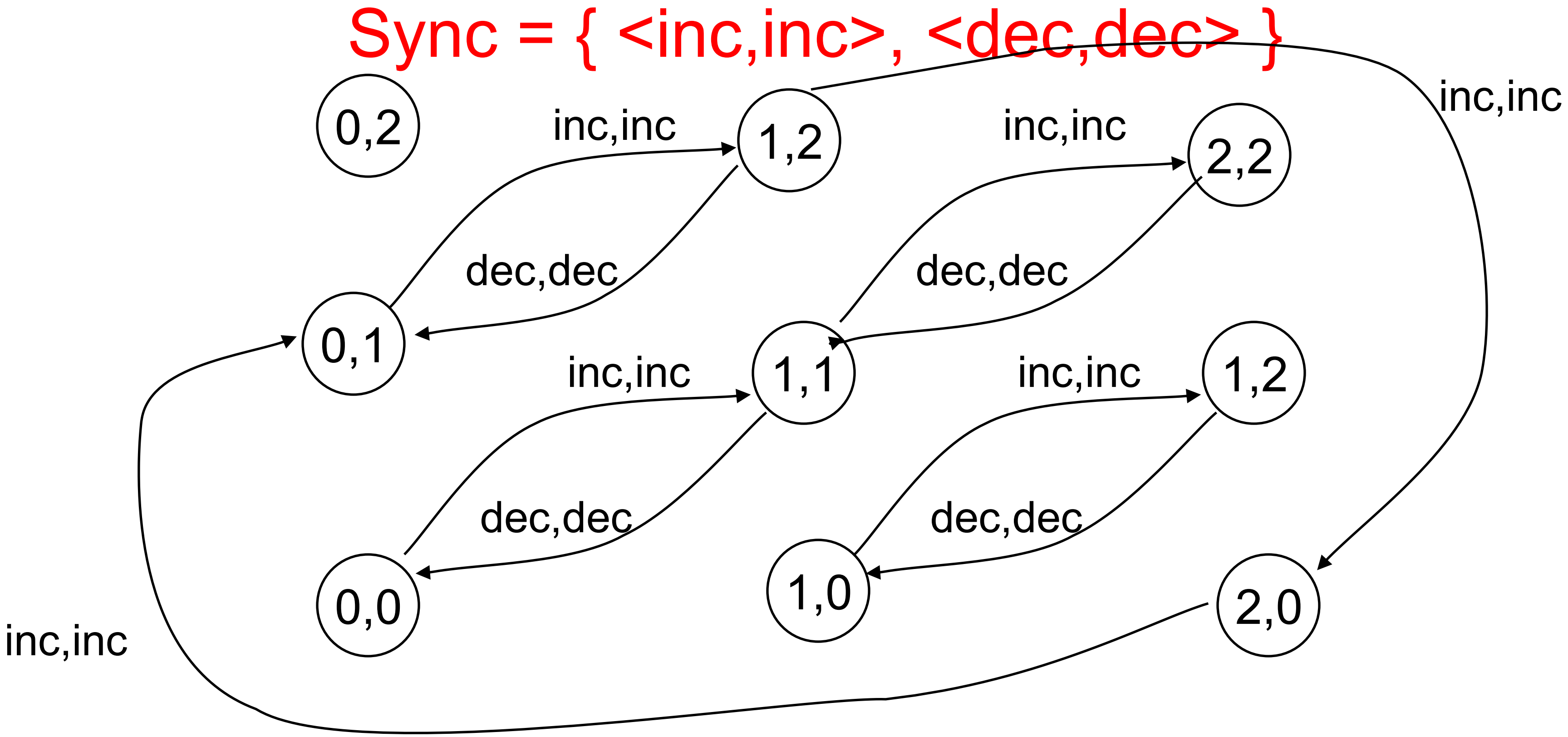
- Subsystems interact by doing some step together (synchronization).
- To synchronize subsystems we restrict the transitions allowed in their cartesian product.
- A synchronization set

  $$\text{Sync} \subseteq L1 \times \ldots \times Ln$$

  define the labels of those transitions corresponding to a synchronization. Transitions with other labels are forbidden.

Example

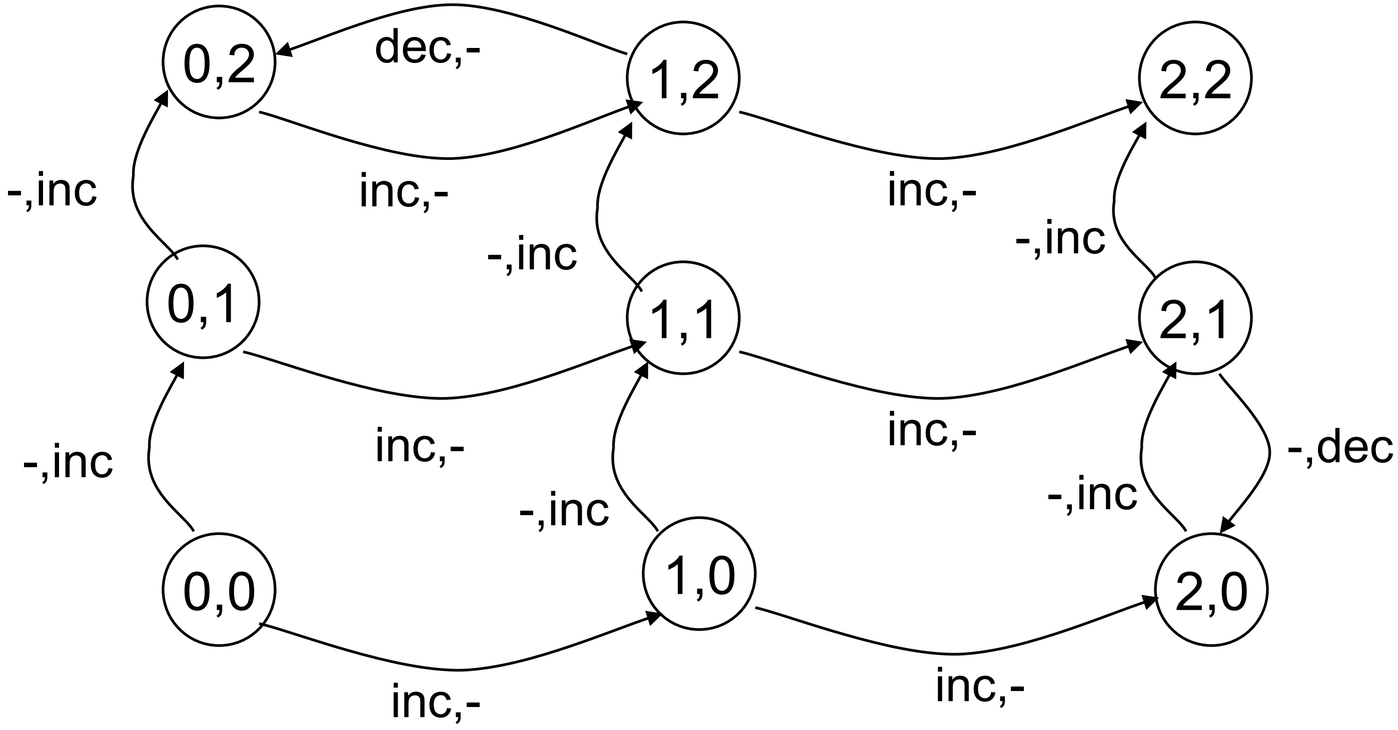
- Few transitions of two counters counting at the same time



Example

- Few transitions of two counters counting one at the time

Sync = { <inc,->, <dec,->, <-,inc>, <-,dec>}



Message Passing

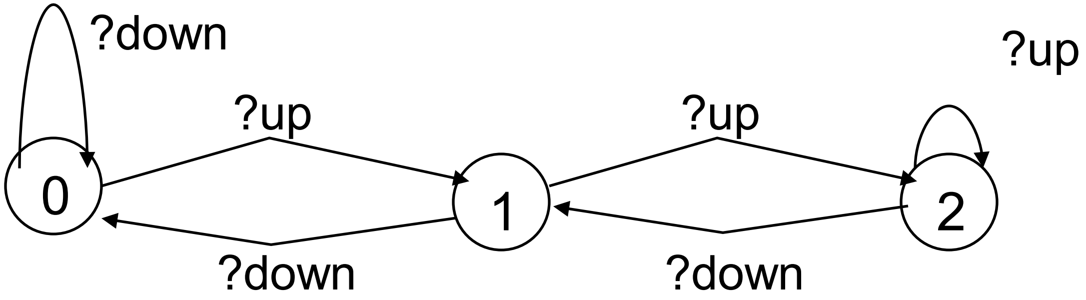
- A special case of synchronized product
- Two special sets of labels
  - !m emission of message m
  - ?m reception of message m
- In message passing, only transitions in which a given emission is executed simultaneously with the corresponding reception will be permitted

Example: An elevator

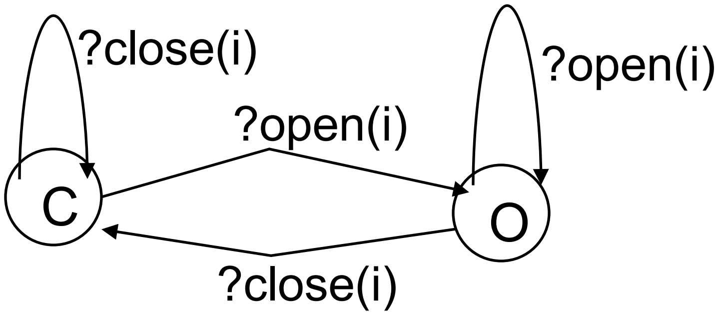
- An elevator in a three floors building consists of
  - ☐ a cabin which goes up and down
  - ☐ three doors which open an close
  - ☐ a controller which commands the three doors and the cabin
- Elevator requests from people at one of the three floors are not modeled, as they are the environment outside the system

Example: An elevator

- The cabin

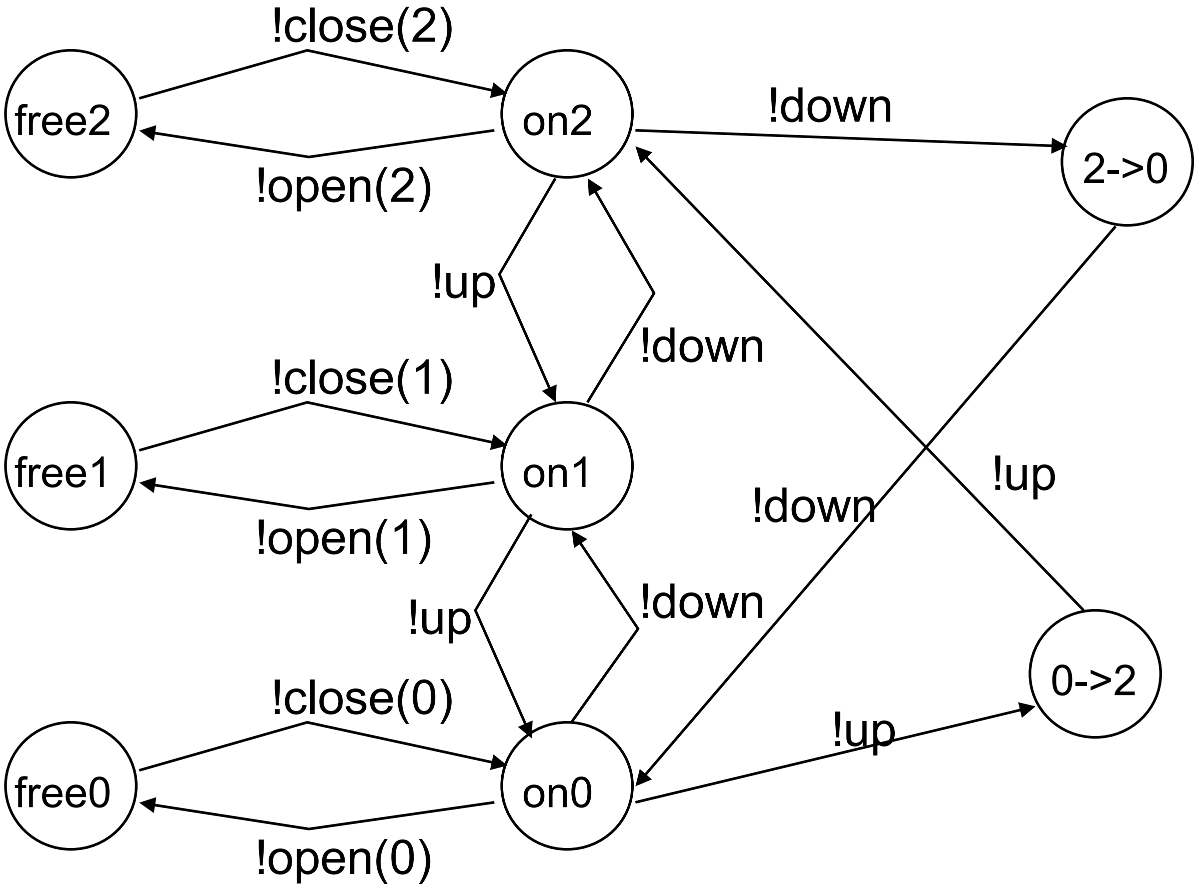


- The i-th door



Example: An elevator

- The controller



Example: An elevator

- The synchronization
  - ☐ Sync =

{<?open(0),-,-,-,!open(0)>,<?close(0),-,-,-,!close(0)>,

<-,?open(1),-,-,!open(1)>,<-,?close(1),-,-,!close(1)>,

<-,-,?open(2),-,!open(2)>,<-,-,?close(2),-,!close(2)>,

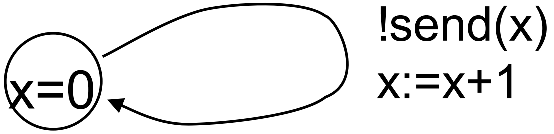
<-,-,-,?down,!down>,<-,-,-,?up,!up>}

Asynchronous Messages

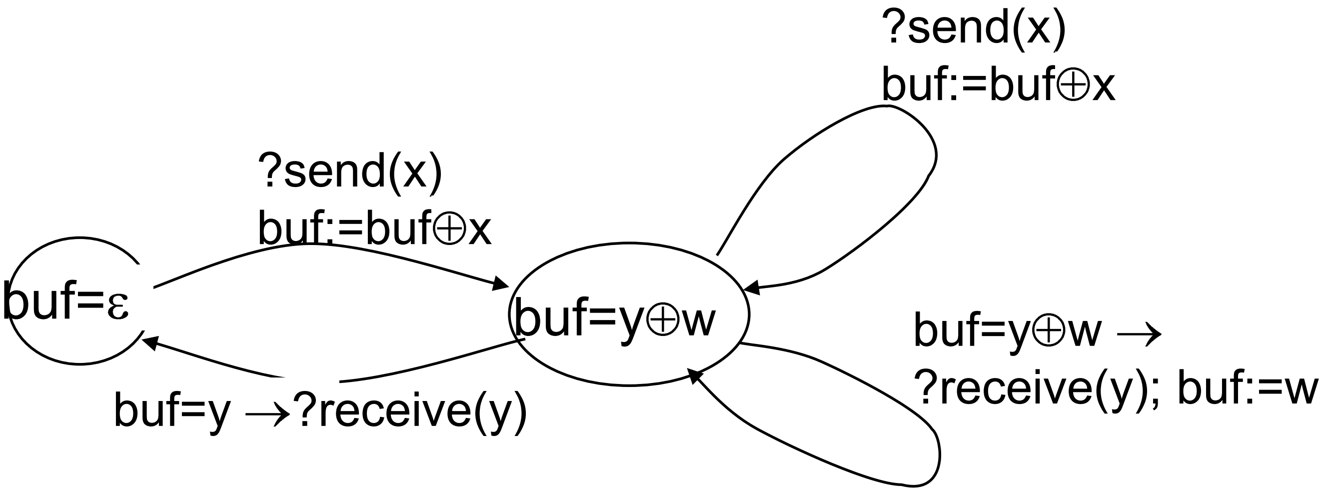
- Like message passing, but messages are not received instantly.
- Emitted messages but not yet received remain in a communication channel, usually a FIFO buffer
- A communication channel can be modeled by a transition system with a variable (for the buffer content)

Example:

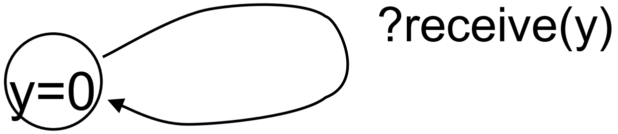
- Producer



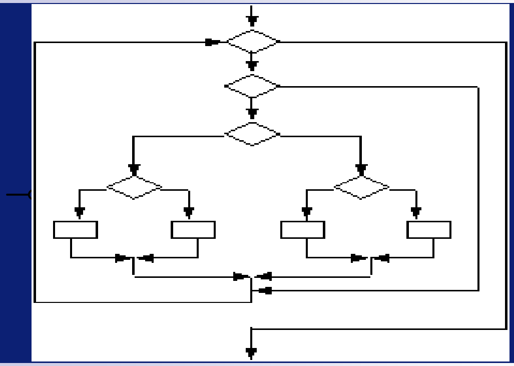
- Buffer



- Consumer



Spring 2007



Program correctness

Linear Time Temporal Logic

Marcello Bonsangue

Formal Verification

Verification techniques comprise

- a modelling framework $\mathcal{M}$

  to describe a system

- a specification language ϕ

  to describe the properties to be verified

- a verification method $\mathcal{M} \vDash \phi, \quad \Gamma \vdash \phi$

  to establish whether a model satisfies a property

Motivations

- For an elevator system, consider the requirements:
  - any request must ultimately be satisfied
  - the elevator never traverses a floor for which a request is pending without satisfying it

- Both concern the dynamic behavior of the system. They can be formalized using a time-dependent notation, like

$$z(t) = 1/2gt^2$$

for the free-falling elevator

Example

- In first order logic, with
  - E(t) = elevator position at time t
  - P(n,t) = pending request at floor n at time t
  - S(n,t) = servicing of floor n at time t

Any request must ultimately be satisfied

$$\forall t\ \forall n\ (\ P(n,t) \Rightarrow \exists t' > t : S(n,t')\ )$$

The elevator never traverse a floor for which a request is pending without satisfying it

$$\forall t \forall t'>t \forall n\ (P(n,t) \wedge E(t') \neq n \wedge \exists t < t'' < t' : E(t'')=n) \Rightarrow \exists t < t''' < t' : S(n,t''')$$

Temporal Logic

- First order logic is too cumbersome for these specifications

- Temporal logic is a logic tailored for describing properties involving time
  - ☐ the time parameter t disappears
  - ☐ temporal operators mimic linguistic constructs
    - always, until, eventually
  - ☐ the truth of a proposition depend on the state on which the system is

LTL: the language

- Atomic propositions $p_1, p_2, \ldots, q, \ldots$
  - to make statements about states of the system
  - elementary descriptions which in a given state of the system have a well-defined truth value:
    - the printer is busy
    - nice weather
    - open
    - x+2=y
  - Their choice depend on the system considered

LTL: the language

- Boolean combinators

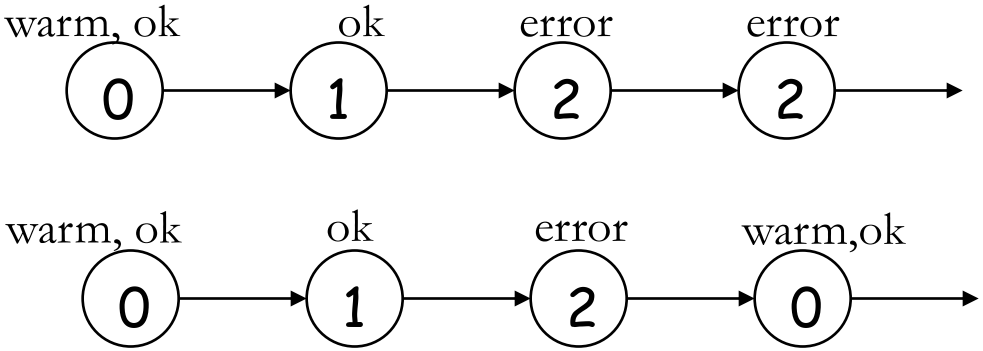
☐ true	$\top$
☐ false	$\bot$
☐ negation	$\neg$
☐ conjunction	$\wedge$
☐ disjunction	$\vee$
☐ implication	$\Rightarrow$

Note: read p $\Rightarrow$ q as "*if p then q*" rather than "*p implies q*".
Try (1 = 2) $\Rightarrow$ Sint_Klas_exists

LTL: the language

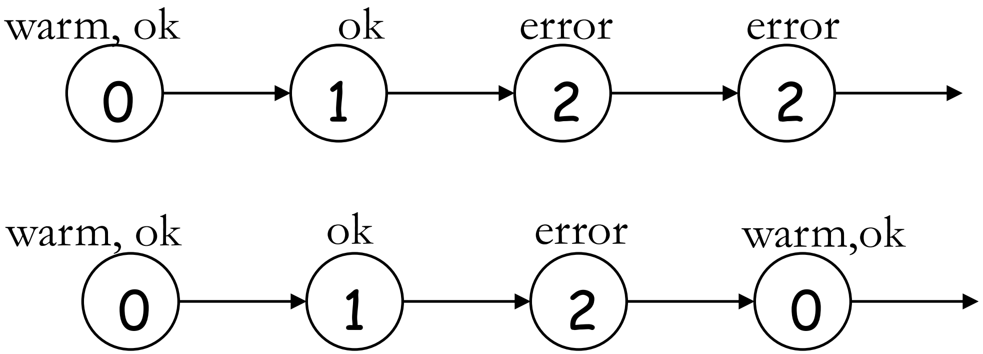
- Temporal combinators allows to speak about the sequencing of states along a computation (rather than about states individually)

- neXt X
  - ☐ Xϕ = *in the next state ϕ holds*
  - ☐ Examples: XXerror and XXXok



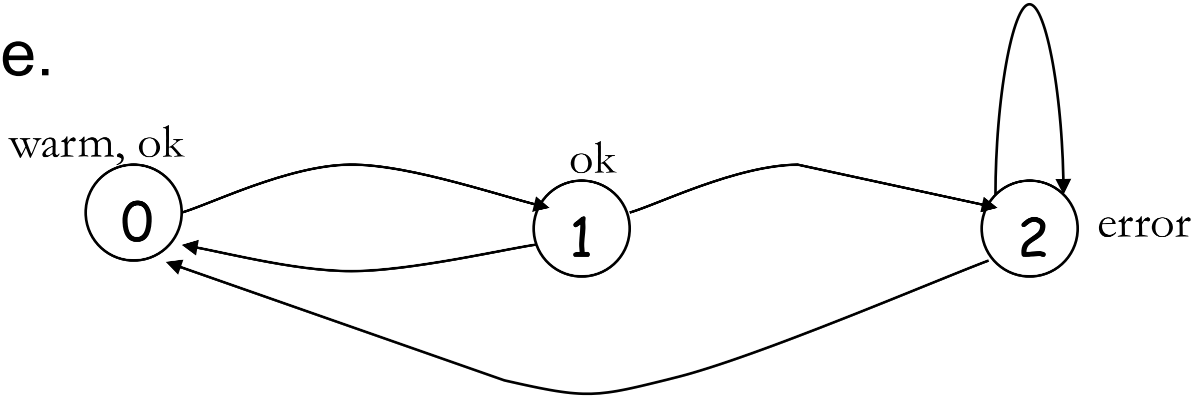
LTL: Temporal combinators

- Future F
  - $F\phi$ = *in some future state ϕ holds* (at least once and without saying in which state)
  - For example, warm ⇒ Fok holds if we are in a “warm” state then we will be in an “ok” state.



LTL: Temporal combinators

- Globally G
  - ☐ $G\phi$ = in all future states ϕ always holds
  - ☐ It is the dual of F: $G\phi = \neg F \neg \phi$
  - ☐ For example $G(\text{warm} \Rightarrow F\text{ok})$ holds if at any time when we are in a “warm” state we will later be in an “ok” state.



  - ☐ $G(\text{warm} \Rightarrow X\neg\text{warm})$? $G(\text{ok} \Rightarrow X\text{warm})$?

LTL: Temporal combinators

- Until U
  - $\phi_1 U \phi_2$ = ϕ_2 will hold in some future state, and in all intermediate states ϕ_1 will hold.
- Weak until W
  - $\phi_1 W \phi_2$ = ϕ_1 holds in all future states until ϕ_2 holds
  - it may be the case ϕ_2 will never hold

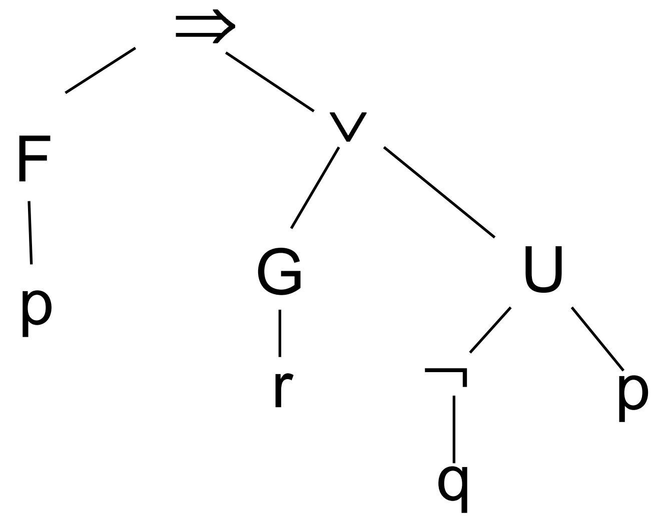
LTL: Temporal combinators

- Release R
  - $\phi_1 R \phi_2$ = ϕ_2 holds in all future state up to (and including) a state when ϕ_1 holds (if ever).
  - It is the dual of U: $\phi_1 R \phi_2 = \neg(\neg\phi_1 U \neg\phi_2)$

LTL - Priorities

- Unary connectives bind most tightly
  - ¬, X,F,G
- Next come U, R and W
- Finally come ∧, ∨ and ⇒

$Fp \Rightarrow Gr \vee \neg qUp$

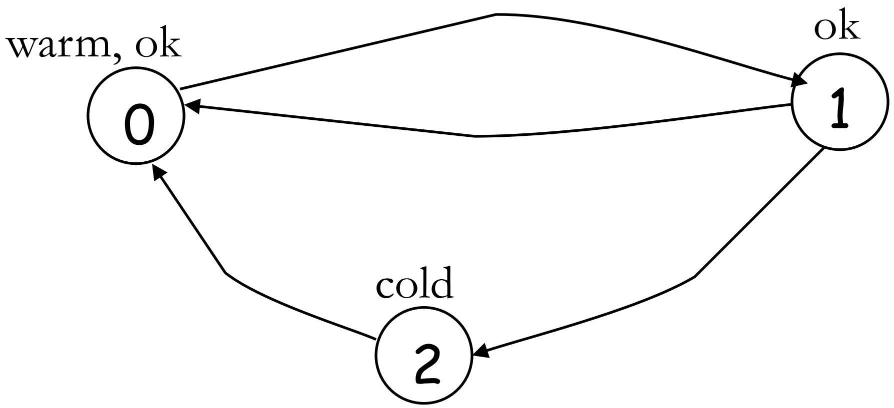


LTL models: Transition Systems

- Transition system: <S,→,L>
  - ☐ S — set of states
  - ☐ L:S → $\mathcal{P}$(Atoms) — labelling function
  - ☐ → ⊆ SxS — transition relation
  - ☐ Every state s has some successor state s' with s → s'

- A system evolves from one state to another under the action of a transition

- We label a state with propositions that hold in that state



Computation paths

- Path: an infinite sequence π of states such that each consecutive pair is connected by a transition

$$0 \to 1 \to 2 \to 0 \to \ldots$$

- For i ≥ 1, we write π^i for the suffix of a path π starting at i.

Semantics (I)

- Let $M = \langle S, \rightarrow, L \rangle$ be a transition system, and $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ a path of M.

- $\pi \models \top$ always
- $\pi \models p$ iff $p \in l(s_1)$
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$

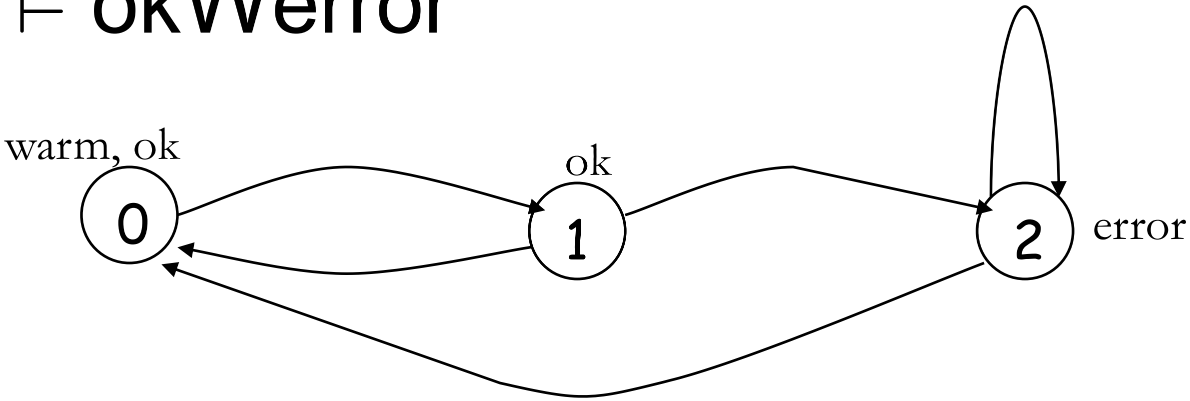
Semantics (II)

- $\pi \models X\phi$ iff $\pi^2 \models \phi$
- $\pi \models F\phi$ iff there is $1 \leq i$ such that $\pi^i \models \phi$
- $\pi \models G\phi$ iff for all $1 \leq i$, $\pi^i \models \phi$
- $\pi \models \phi_1 U \phi_2$ iff there is $1 \leq i$ such that $\pi^i \models \phi_2$
  and for all $j<i$, $\pi^j \models \phi_1$
- $\pi \models \phi_1 W \phi_2$ iff either $\pi \models \phi_1 U \phi_2$ or for all $1 \leq i$, $\pi^i \models \phi_2$
- $\pi \models \phi_1 R \phi_2$ iff either there is $1 \leq i$ such that $\pi^i \models \phi_1$
  and for all $j \leq i$, $\pi^j \models \phi_2$
  or for all $1 \leq k$, $\pi^k \models \phi_2$

System properties

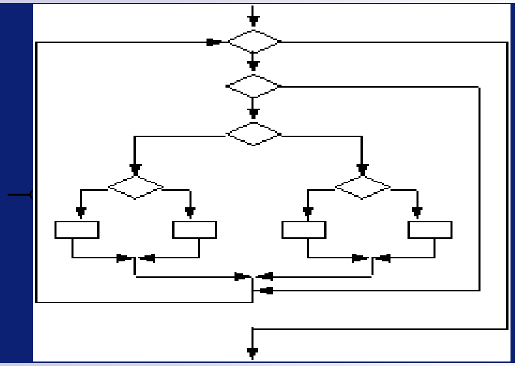
- M,s ⊨ ϕ iff π ⊨ ϕ for every path π of M starting from the state s

- M,0 ⊨ okWerror



- M,0 ⊭ okUerror (Why?)

Spring 2007



Program correctness

Using temporal logics

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

LTL equivalences

- De Morgan-based
  - $\neg F\phi \equiv G\neg\phi$
  - $\neg X\phi \equiv X\neg\phi$

X-self duality: on a path each
state has a unique successor

- Until reduction
  - $F\phi \equiv T \; U \; \phi$
  - $F\phi \equiv T \; U \; \phi$

LTL: Adequate sets of connectives

- Theorem: The set of operators

  $$T, \neg, \wedge, U, X$$

  is adequate for LTL.

  - ☐ $\phi U \psi \equiv \neg(E[\neg\psi U(\neg\phi \wedge \neg\psi)] \vee AG\, \neg\psi)$

  - ☐ $\phi R \psi \equiv \neg(\neg\phi U\, \neg\psi)$
  - ☐ $\phi W \psi \equiv \psi R(\phi \vee \psi)$

Other LTL equivalences

- $G\phi \equiv \phi \wedge XG\phi$
- $F\phi \equiv \phi \vee XF\phi$
- $\phi U\psi \equiv \psi \vee (\phi \wedge X\phi U\psi)$

- Theorem: $\phi U\psi \equiv \neg(\neg\psi U(\neg\phi \wedge \neg\psi)) \wedge F\psi$

Verification goals

- Formulating properties requires some expertise
- Today we present categories of fundamental properties commonly used for system verification
    - reachability properties
    - safety properties
    - liveness properties
    - fairness properties

Reachability

- A reachability property states that some particular situation can be reached
  - Simple
    - “We can obtain n < 0”
    - “We can enter a critical section”
  - Conditional
    - “We can enter a critical section without traversing n= 0”
  - Any
    - “we can always return to the initial state”

Reachability in LTL

- LTL misses the existential quantifier on paths, thus it can only express reachability negatively:

something is not reachable

- Simple reachability
  - ¬G(n ≥0)
  - ¬G(no_critic_sec)

Safety

- A safety property states that, under certain conditions, an event never occurs
  - “Two processes will never be both in their critical section”
  - “A memory overflow will never occur”
- In general, safety statements express that an undesirable event will not occur.
- The negation of a reachability property is a safety property (and the other way around)

Safety in LTL

- Typically expressed by the combinator G in LTL
- Examples
  - $G(\neg \text{critic\_sec}_1 \wedge \neg \text{critic\_sec}_2)$
  - $G(\neg \text{overflow})$

- Conditional safety

  "As long the key is not in, the car won't start"
  - $\neg$start W key
  - $\neg$start U key as we are not required to have the key in some day

Liveness

- A liveness property states that, under certain conditions, an event will ultimately occur
  - “Any request will be satisfied”
  - “The light will turn green”
  - “after the rain, the sunshine”

- Liveness is not reachability

  “The light will turn green (some day, regardless of the system behavior)”

  vs.

  “It is possible for the light (some day) to turn green”

Liveness

- In general, liveness statements express that happy event will occur in the end

- Termination is a liveness property:
  - “The program will terminate”

Liveness in LTL

- Typically expressed by the combinator F
- Examples
    - $G(req \Rightarrow F sat)$ in LTL
- In LTL $\phi_1 U \phi_2$ is a liveness property, whereas $\phi_1 W \phi_2$ is a safety property

Deadlock

- A deadlock property states that, the system can never be in a situation in which no progress is possible

- Safety? Liveness?
  - ☐ Deadlock freeness in LTL

    GX T

  whatever state may be reached (G) there exists an immediate successor state (X T)

Fairness

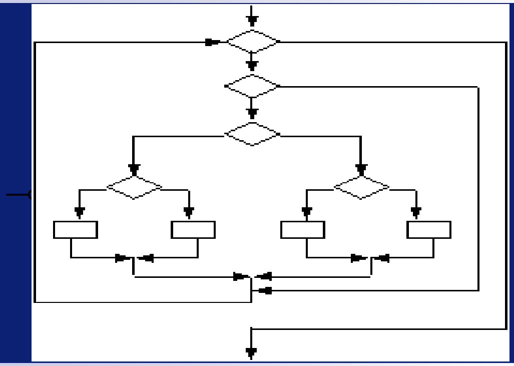
- A fairness property states that, under certain conditions, an event will occur (or will fail to occur) infinitely often
    - "If access to a critical section is infinitely often requested, then access will be granted infinitely often

Fairness in LTL

- Typically expressed by the combinators
  - GF (infinitely often)
  - FG (eventually always)

- Examples
  - GF critic_in $\vee$ FG$\neg$ critic_req

Spring 2007



Program correctness

LTL equivalences

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

LTL equivalences

- We say that two LTL formulas ϕ and ψ are semantically equivalent, writing $\phi \equiv \psi$ if for all models M and for all paths π of M we have

$$\pi \vDash \phi \text{ iff } \pi \vDash \psi$$

De Morgan-based equivalences

- $\neg F\phi \equiv G\neg\phi$
- $\neg G\phi \equiv F\neg\phi$
- $\neg X\phi \equiv X\neg\phi$

X-self duality: on a path each state has a unique successor

- $\neg(\phi \; U \; \psi) \equiv \neg\phi \; R \; \neg\psi$
- $\neg(\phi \; R \; \psi) \equiv \neg\phi \; U \; \neg\psi$

Distributivities

- $F(\phi \vee \psi) \equiv F\phi \vee F\psi$
- $G(\phi \wedge \psi) \equiv G\phi \wedge G\psi$

Reductions

- $F\phi \equiv T \; U \; \phi$
- $G\phi \equiv \bot \; R \; \phi$

- $\phi \; U \; \psi \equiv \phi \; W \; \psi \wedge F\psi$
- $\phi \; W \; \psi \equiv \phi \; U \; \psi \vee F\psi$

- $\phi \; W \; \psi \equiv \psi \; R \; (\phi \vee \psi)$
- $\phi \; R \; \psi \equiv \psi \; W \; (\phi \wedge \psi)$

LTL: Adequate sets of connectives

- A set of operators S is adequate for LTL if every formula in LTL can be expressed as an equivalent one using only the operators in S.

- <u>Theorem</u>: The set of operators

  $$\top, \neg, \wedge, X, U$$

  is adequate for LTL.

- Without negation, the set of operators

  $$\top, \bot, \vee, \wedge, X, U, R$$

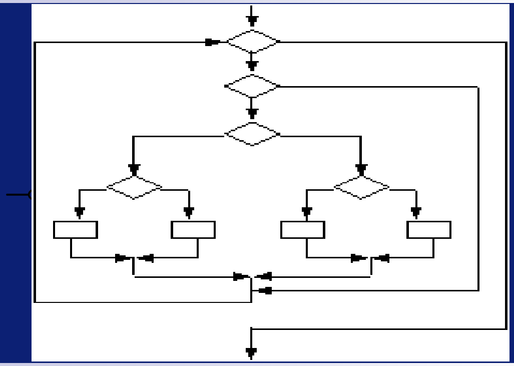
  is adequate but $\top, \bot, \vee, \wedge, X, R, G$ is not (because one cannot define F).

Other LTL equivalences

- $G\phi \equiv \phi \wedge XG\phi$
- $F\phi \equiv \phi \vee XF\phi$
- $\phi U\psi \equiv \psi \vee (\phi \wedge X(\phi U\psi))$

- <u>Theorem</u>: $\phi U\psi \equiv \neg(\neg\psi U(\neg\phi \wedge \neg\psi)) \wedge F\psi$

Spring 2007



Program correctness

Branching-time temporal logics

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

CTL

- CTL = Computational Tree Logic
  - ☐ the temporal combinators are under the immediate scope of the path quantifiers

- Why CTL? The truth of CTL formulas depends only on the current state and not on the current execution!

<u>Benefit</u>: easy and efficient model checking

<u>Disadvantages</u>: hard for describing individual path

The language

- Path quantifiers allows to speaks about sets of executions.
  - ☐ The model of time is tree-like: many futures are possible from a given state
- Inevitably $A\phi$

  from the current state all executions satisfy ϕ
- Possibly $E\phi$

  from the current state there exists an execution satisfying ϕ

CTL - Syntax

- $\phi ::= p_1 \mid p_2 \mid \ldots$

  $\top \mid \bot \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid$

  $AX\phi \mid AF\phi \mid AG\phi \mid A[\phi\ U\ \phi] \mid$

  $EX\phi \mid EF\phi \mid EG\phi \mid E[\phi\ U\ \phi]\ .$

CTL - Priorities

- Unary connectives bind most tightly
  - $\neg$, AG, EG, AF, EF, AX, and EX
- Next come $\wedge$, and $\vee$
- Finally come, AU and EU

- Example:

  $AGp_1 \Rightarrow EGp_2$ is not the same as $AG(p_1 \Rightarrow EGp_2)$

CTL - yes or no?

- Yes
  - $EFE[p \; U \; q]$
  - $A[p \; U \; EF \; q]$
- No
  - $EF(p \; U \; q)$
  - $FG \; p$
- Yes or no?
  - $AG(p \Rightarrow A[p \; U \; (\neg p \wedge A[\neg p \; U \; q])])$
  - $AF[(p \; U \; q) \wedge (q \; U \; p)]$

A is not G

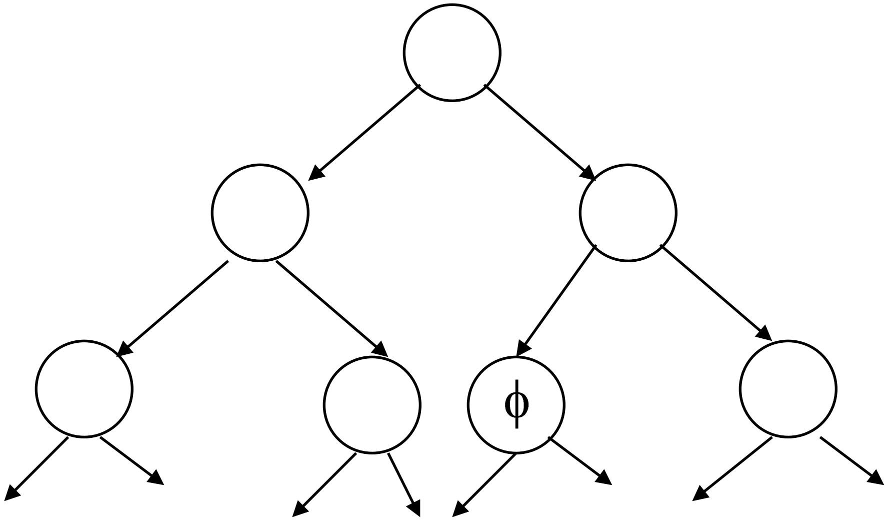
- Aϕ states that all the executions starting from the current state will satisfy ϕ
- Gϕ state that ϕ holds at every state of the execution considered

- A and E quantify over paths in a tree
- G and F quantify over positions along a given path in a tree

Combining E and F (I)

- EFϕ

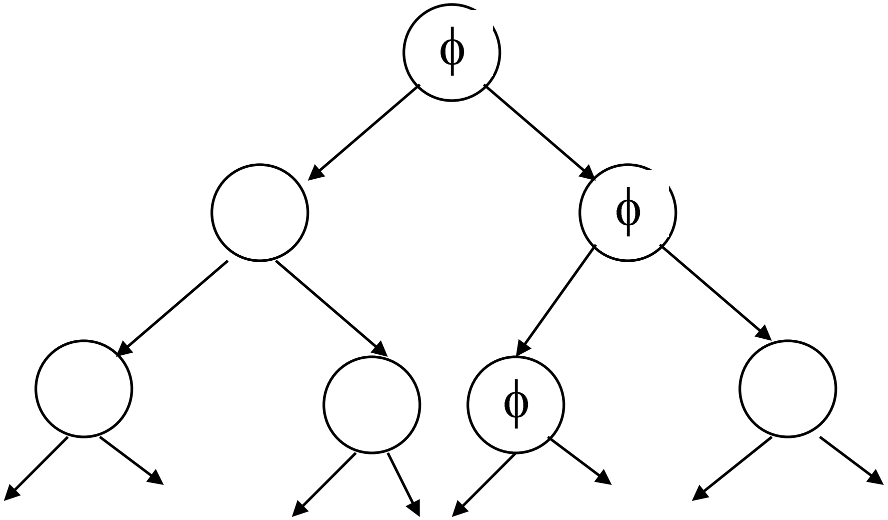
  “it is possible that ϕ will hold in the future”



Combining E and F (II)

- $EG\phi = E\neg F\neg\phi$

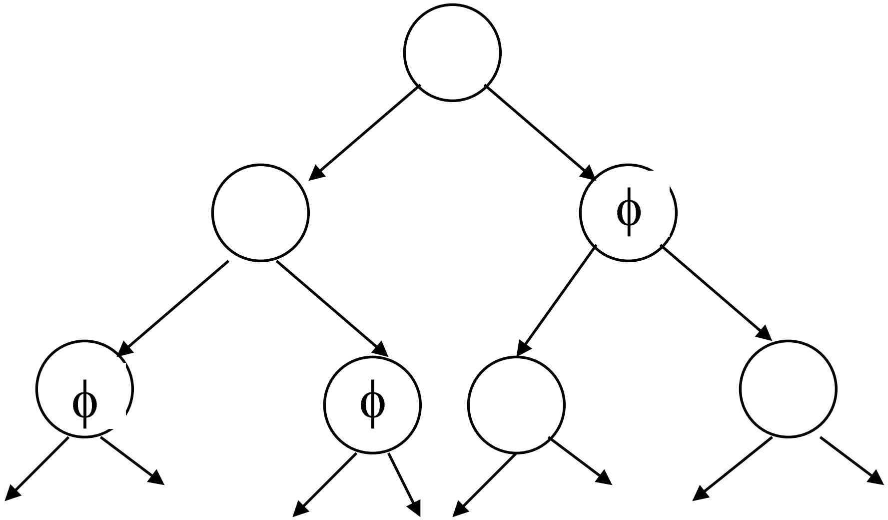
  "it is possible that ϕ will always hold"



Combining E and F (III)

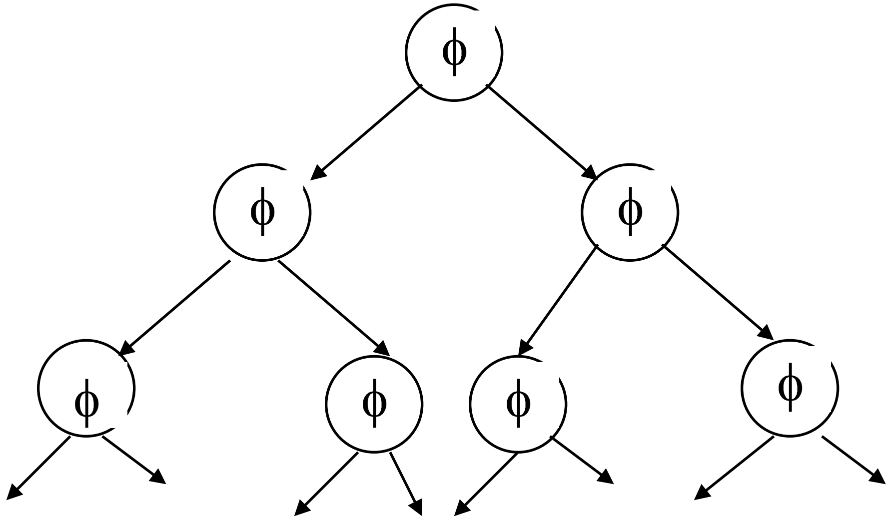
- $AF\phi = \neg E \neg F \phi$

  "it is inevitable that ϕ will hold in the future"



Combining E and F (IV)

- $AG\phi = \neg EF \neg \phi$
  "ϕ is always true"



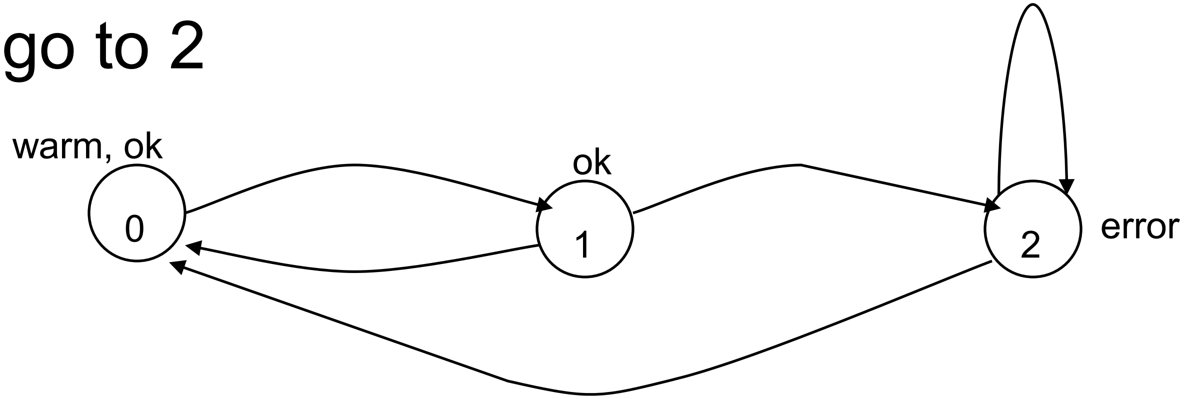
- In this case ϕ is an invariant, that is, a property that is true continuously

Example

- All executions starting from 0 satisfy

  AFEXerror

  Why? Because from 0 all executions traverse 1 and may go to 2



- There exists an execution which does not satisfy AFAXerror. Which one?

Examples

- AGEFϕ

Along every execution (A)
from every state (G)
it is possible (E)
that we will encounter a state (F)
satisfying ϕ

that is, ϕ is always reachable

CTL - Satisfaction

- Let $M = <S,\rightarrow,I>$ be a transition system with $I(s)$ the set of atomic propositions satisfied by a state $s \in S$.

- <u>Idea for a model</u>: A CTL formula refers to a given state of a given transition system
    - $M,s \vDash \phi$ means "ϕ is true at state s"

We will define it by induction
on the structure of ϕ

CTL - Semantics (I)

- $M,s \models T$ for all s in S
- $M,s \models p$ iff $p \in I(s)$
- $M,s \models \neg\phi$ iff $\models$ not $M,s \models \phi$
- $M,s \models \phi_1 \wedge \phi_2$ iff $M,s \models \phi_1$ and $M,s \models \phi_2$

:

:

CTL - Semantics (II)

- $M,s \models AX\phi$ iff for all s' such that $s \to s'$ we have $M,s' \models \phi$
- $M,s \models EX\phi$ iff there exists s' such that $s \to s'$ and $M,s' \models \phi$

CTL - Semantics (III)

- $M,s \vDash AG\phi$ iff for all executions

  $s_0 \to s_1 \to s_2 \to s_3 \ldots$ with $s = s_0$ we have $M,s_i \vDash \phi$

- $M,s \vDash EG\phi$ iff there exists an execution

  $s_0 \to s_1 \to s_2 \to s_3 \ldots$ with $s = s_0$ and such that $M,s_i \vDash \phi$

CTL - Semantics (IV)

- $M,s \vDash AF\phi$ iff for all executions $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \ldots$ with $s = s_0$ there is i such that $M,s_i \vDash \phi$
- $M,s \vDash EF\phi$ iff there exists an execution $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \ldots$ with $s=s_0$ and there is i such that $M,s_i \vDash \phi$

CTL - Semantics (V)

- $M,s \vDash A[\phi_1 U \phi_2]$ iff for all executions $s \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \ldots$ there is i such that $M,s_i \vDash \phi_2$ and for each $j < i$ $M,s_j \vDash \phi_1$

- $M,s \vDash E[\phi_1 U \phi_2]$ iff there exists an execution $s \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \ldots$ and there is i such that $M,s_i \vDash \phi_2$ and for each $j < i$ $M,s_j \vDash \phi_1$

CTL equivalences

- De Morgan-based
  - $\neg AF\phi \equiv EG\neg\phi$
  - $\neg EF\phi \equiv AG\neg\phi$
  - $\neg AX\phi \equiv EX\neg\phi$

X-self duality: on a path each state has a unique successor

- Until reduction
  - $AF\phi \equiv A[T \, U \, \phi]$
  - $EF\phi \equiv E[T \, U \, \phi]$

CTL: Adequate sets of connectives

- Theorem: The set of operators

  $\top, \neg, \wedge$, {AX or EX}, {EG,AF or AU}, and EU

  is adequate for CTL.

  - $A[\phi U \psi] \equiv \neg(E[\neg\psi U(\neg\phi \wedge \neg\psi)] \vee EG\,\neg\psi)$

CTL: Weak until and release

- Use LTL equivalence to define:
  - $A[\phi R \psi] \equiv \neg E[\neg\phi U \neg\psi]$
  - $E[\phi R \psi] \equiv \neg A[\neg\phi U \neg\psi]$

  - $A[\phi W \psi] \equiv A[\psi R(\phi \vee \psi)]$
  - $E[\phi W \psi] \equiv E[\psi R(\phi \vee \psi)]$

Other CTL equivalences

- $EG\phi \equiv \phi \wedge EX\ EG\phi$
- $AG\phi \equiv \phi \wedge AX\ AG\phi$

- $AF\phi \equiv \phi \vee AX\ AF\phi$
- $EF\phi \equiv \phi \vee EX\ EF\phi$

- $A[\phi U\psi] \equiv \psi \vee (\phi \wedge AXA[\phi U\psi])$
- $E[\phi U\psi] \equiv \psi \vee (\phi \wedge EXE[\phi U\psi])$

CTL* - Syntax

- State formulas (evaluated in states)

$$\phi ::= T \mid p \mid \neg\phi \mid \phi \wedge \phi \mid A\psi \mid E\psi$$

- Path formulas (evaluated along paths)

$$\psi ::= \phi \mid \neg \psi \mid \psi \wedge \psi \mid X\psi \mid F\psi \mid G\psi \mid \psi\, U\psi$$

Examples

- AGFϕ

Along every execution (A)
from every state (G)
we will encounter a state (F)
satisfying ϕ

that is, ϕ is satisfied infinitely often

Model

- Let M = <S,→,l> be a transition system with l(s) the set of atomic propositions satisfied by a state s ∈S.

- Idea for a model: A formula of temporal logic refers to an instant i of an execution π of a transition system M

- $M,\pi,i \vDash \phi$ means

  "ϕ is true at position i of path π of M"

Semantics (I)

- $M,\pi,i \models T$ always
- $M,\pi,i \models p$ iff $p \in l(\pi(i))$
- $M,\pi,i \models \neg\phi$ iff not $M,\pi,i \models \phi$
- $M,\pi,i \models \phi_1 \wedge \phi_2$ iff $M,\pi,i \models \phi_1$ and $M,\pi,i \models \phi_2$

Semantics (II)

- $M,\pi,i \models X\phi$ iff $M,\pi,i+1 \models \phi$
- $M,\pi,i \models F\phi$ iff there exists $i \leq j$ such that $M,\pi,j \models \phi$
- $M,\pi,i \models G\phi$ iff $M,\pi,j \models \phi$ for all $i \leq j$
- $M,\pi,i \models \phi_1 U \phi_2$ iff there exists $i \leq j$ such that $M,\pi,j \models \phi_2$ and for all $i \leq k < j$ we have $M,\pi,k \models \phi_1$

Semantics (III)

- $M,\pi,i \vDash E\phi$ iff there exists π' such that $\pi(0)\ldots\ \pi(i) = \pi'(0)\ldots\ \pi'(i)$ and $M,\pi',i \vDash \phi$

- $M,\pi,i \vDash A\phi$ iff for all π' such that $\pi(0)\ldots\ \pi(i) = \pi'(0)\ldots\ \pi'(i)$ we have $M,\pi',i \vDash \phi$

LTL and CTL $\subseteq$ CTL*

- Semantically, an LTL formula ϕ is equivalent to the CTL* formula $A\phi$

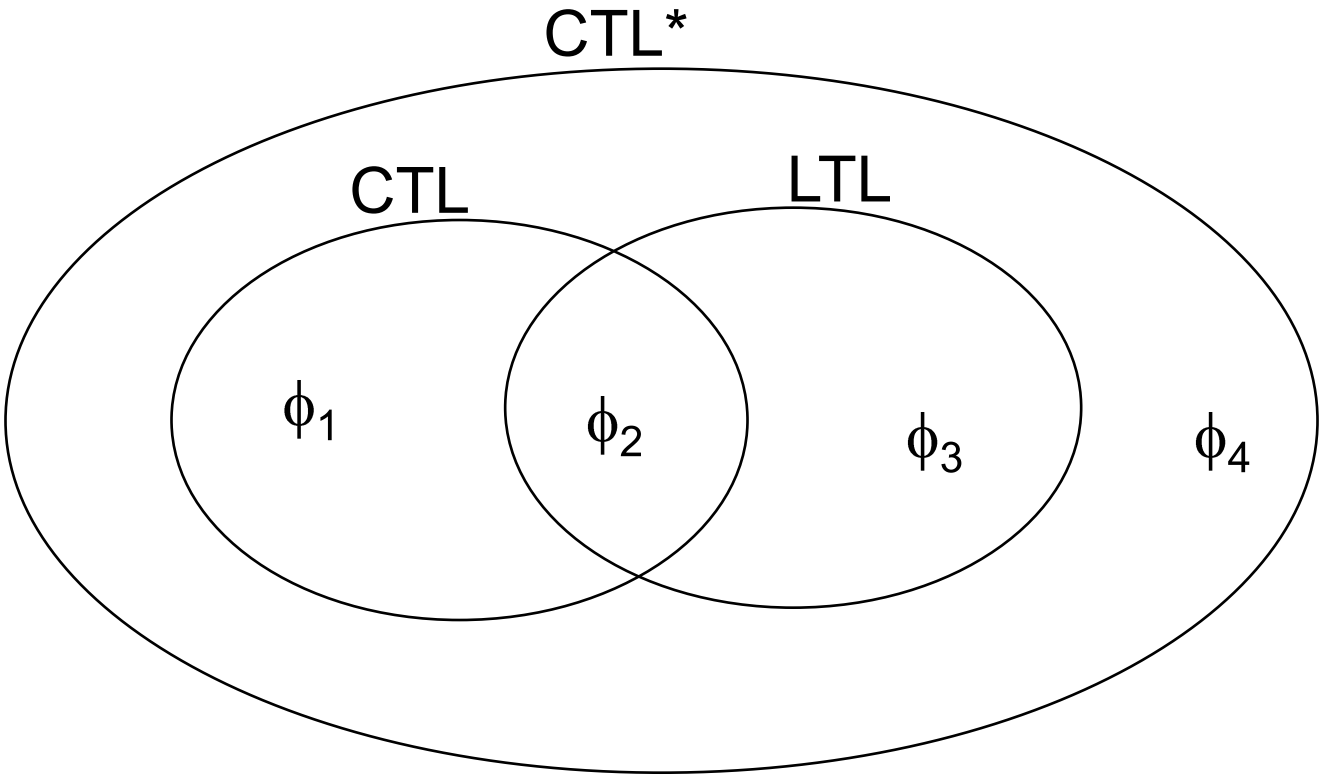
- CTL is a restricted fragment of CTL* with path formulas

$$\psi ::= X\phi \mid F\phi \mid G\phi \mid \phi \; U \; \phi$$

and the same state formulas as CTL*, i.e.

$$\phi ::= T \mid p \mid \neg\phi \mid \phi \wedge \phi \mid A\psi \mid E\psi$$

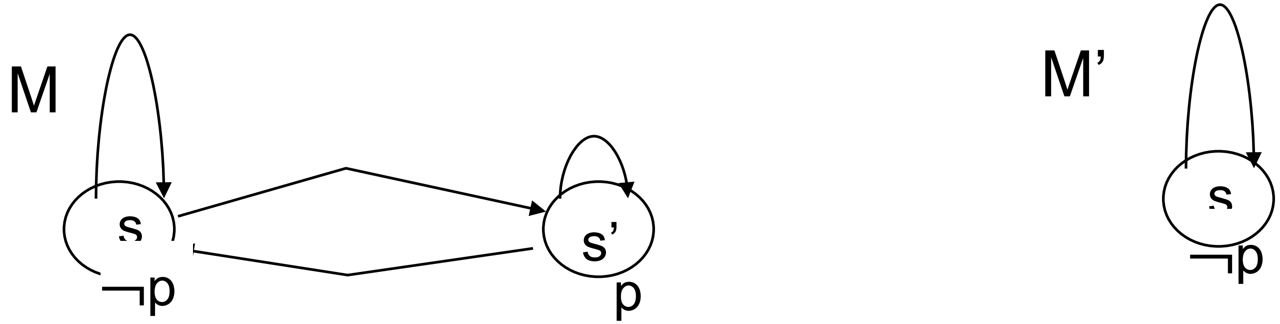
Expressivity



In CTL but not in LTL

ϕ_1 = AG EF p in CTL

From any state we can always get to a state in which p holds



- It cannot be expressed as LTL formula ϕ because
  - ☐ All executions starting from s in M’ are also executions starting from s in M
  - ☐ In CTL M,s $\models \phi_1$ but M’,s $\not\models \phi_1$

In CTL and in LTL

$\phi_2 = AG(p \Rightarrow AFq)$ in CTL

and

$\phi_2 = G(p \Rightarrow Fq)$ in LTL

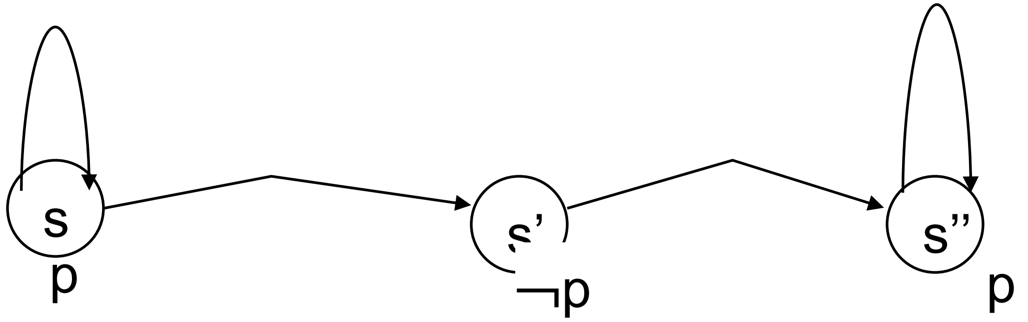
“Any p is eventually followed by a q”

In LTL but not in CTL

ϕ_3 = GFp $\Rightarrow$ Fq in LTL

"If p holds infinitely often along a path, then there is a state in which q holds"

Note: FGp is different from AFAGp since the first is satisfied in



whereas the latter is not (starting from s).

Neither in CTL nor in LTL

$\phi_4 = E(GFp)$ in CTL*

“There is a path with infinitely many state in which p holds”

- Not expressible in LTL: Trivial
- Not expressible in CTL: very complex

Boolean combination of path in CTL

- CTL = CTL* but
  - ☐ Without boolean combination of path formulas
  - ☐ Without nesting of path formulas

- The first restriction is not real …
  - ☐ $E[Fp \wedge Fq] \equiv EF[p \wedge EFq] \vee EF[q \wedge EFp]$
    - First p and then q or viceversa

More generally …

- $E[\neg(pUq)] \equiv E[\neg qU(\neg p \wedge \neg q)] \vee EG\,\neg q$
- $E[(p_1Uq_1) \wedge (p_2Uq_2)] \equiv E[(p_1 \wedge p_2)U(q_1 \wedge E[p_2Uq_2])]$
  $\vee\, E[(p_1 \wedge p_2)U(q_2 \wedge E[p_1Uq_1])]$
- $E[Fp \wedge Gq] \equiv E[q\ U\ (p \wedge EG\ q)]$

- $E[\neg Xp] \equiv EX\neg p$
- $E[Xp \wedge Xq] \equiv EX(p \wedge q)$
- $E[Fp \wedge Xq] \equiv EX(q \wedge EFp)$

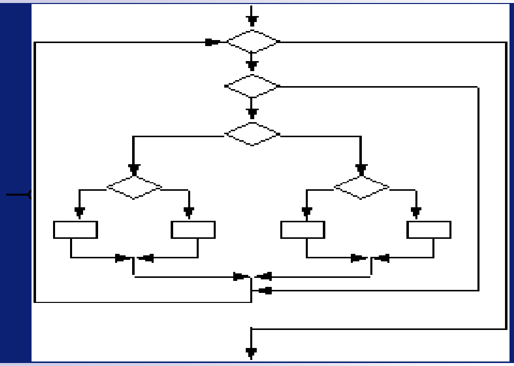
- $A[\phi] \equiv \neg\, E[\neg\, \phi]$

Past operators

		analogues of	
Previous	P	X	neXt
Since	S	U	Until
Once	O	F	Future
Historically	H	G	Globally

- In LTL they do not add expressive power, but CTL they do!

Spring 2007



Program correctness

Model-checking CTL

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

Formal Verification

Verification techniques comprise

- a modelling framework M, Γ

  to describe a system
- a specification language ϕ

  to describe the properties to be verified
- a verification method $M \models \phi, \Gamma \vdash \phi$

  to establish whether a model satisfies a property

Today for CTL

Model Checking

- Question: does a given transition system satisfies a temporal formula?
- Simple answer: use definition of $\models$!
  - We cannot implement it as we have to unwind the transition system in a possibly infinite tree

Can we do better?

and most probably!

The problem

- We need efficient algorithms to solve the problems

  [1] $M,s \overset{?}{\models} \phi$

  [2] $M,\overset{?}{s} \models \phi$

  where M should have finitely many states, and ϕ is a CTL formula.

- We concentrate to solution of [2], as [1] can be easily derived from it.

The solution

- Input: A CTL model M and CTL formula ϕ
- Output: The set of states of M which satisfy ϕ
- Basic principles:
  - ☐ Translate any CTL formula ϕ in terms of the connectives AF, EU,EX, $\wedge$,$\neg$, and $\bot$.
  - ☐ Label the states of M with sub-formulas of ϕ that are satisfied there, starting from the smallest sub-formulas and working outwards towards ϕ
  - ☐ Output the states labeled by ϕ

The labelling

- An immediate sub-formula of a formula ϕ is any maximal-length formula ψ other than ϕ itself
- Let ψ be a sub-formula of ϕ and assume the states of M have been already labeled by all immediate sub-formulas of ψ.
- Which states have to be labeled by ψ?

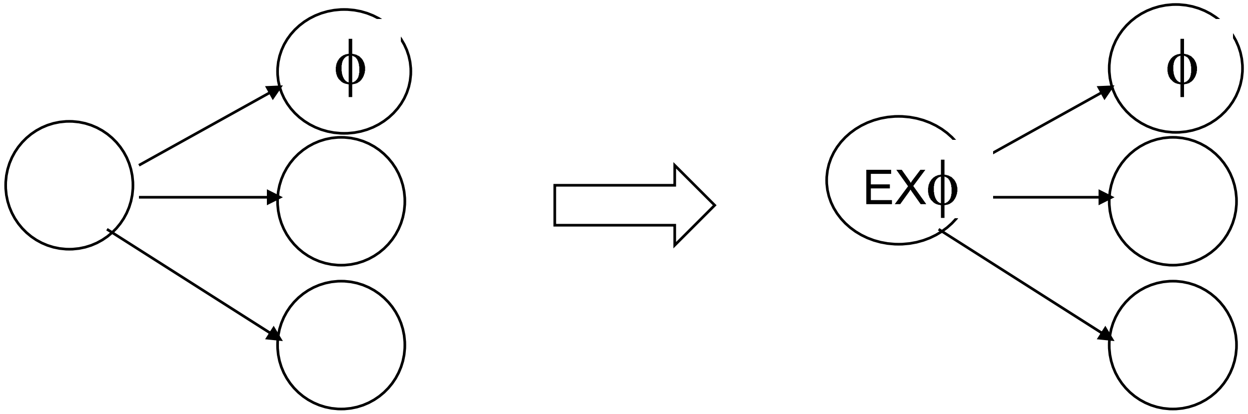
We proceed by case analysis

The basic labeling

$\bot$	no states are labeled
p	label a state s with p if $p \in l(s)$
$\phi_1 \land \phi_2$	label a state s with $\phi_1 \land \phi_2$ if s is already labeled with ϕ_1 and ϕ_2
$\neg\phi$	label a state s with $\neg\phi$ if s is not already labeled with ϕ

The EX labeling

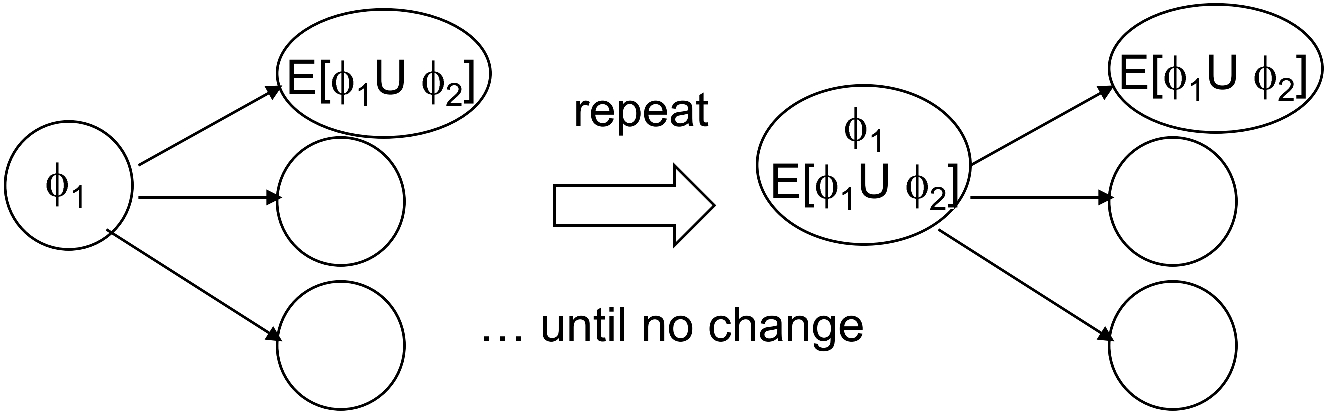
- EXϕ Label with EXϕ any state s with one of its successors already labeled with ϕ



The EU labeling

- $E[\phi_1 U \phi_2] \equiv \phi_2 \vee (\phi_1 \wedge EXE[\phi_1 U \phi_2])$

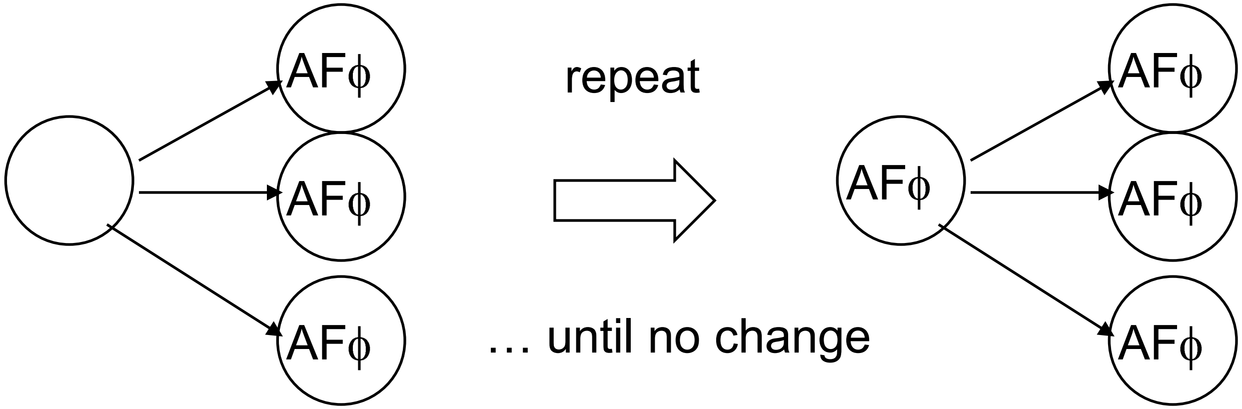
1. Label with $E[\phi_1 U \phi_2]$ any state s already labeled with ϕ_2
2. <u>Repeat until no change</u>: label any state s with $E[\phi_1 U \phi_2]$ if s is labeled with ϕ_1 and at least one of its successor is already labeled with $E[\phi_1 U \phi_2]$



The AF labeling

- $AF\phi \equiv \phi \vee AXAF\phi$

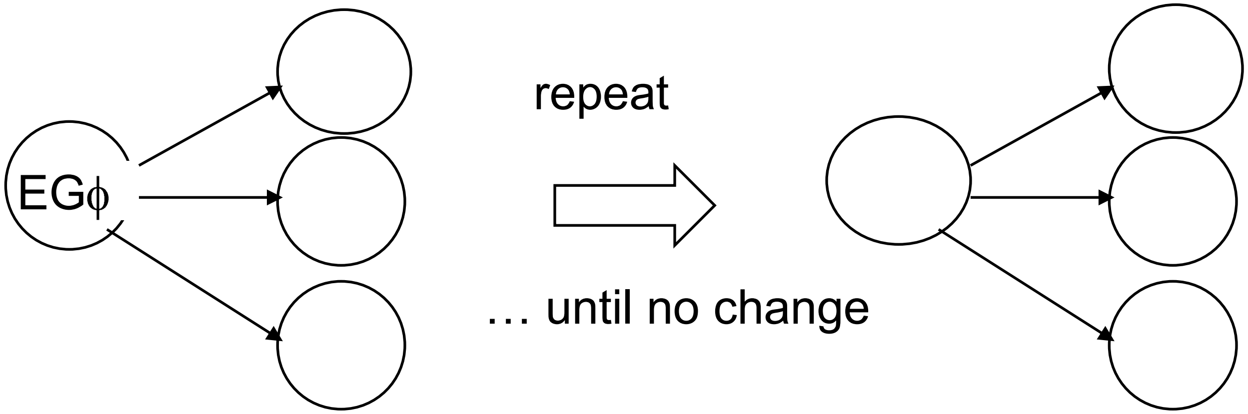
1. Label with $AF\phi$ any state s already labeled with ϕ
2. <u>Repeat until no change</u>: label any state s with $AF\phi$ if all successors of s are already labeled with $AF\phi$



The EG labeling (direct)

- $EG\phi \equiv \phi \wedge EXEG\phi \equiv \neg AF\neg\phi$

1. Label all the states with $EG\phi$
2. Delete the label $EG\phi$ from any state s not labeled with ϕ
3. <u>Repeat until no change</u>: delete the label $EG\phi$ from any state s if none of its successors is labeled with $EG\phi$



Complexity

The complexity of the model checking algorithm is

$$\mathbf{O}(f * V * (V+E))$$

where f = number of connectives in ϕ

V = number of states of M

E = number of transitions of M

It can be easily improved to an algorithm linear both in the size of the formula and of the model

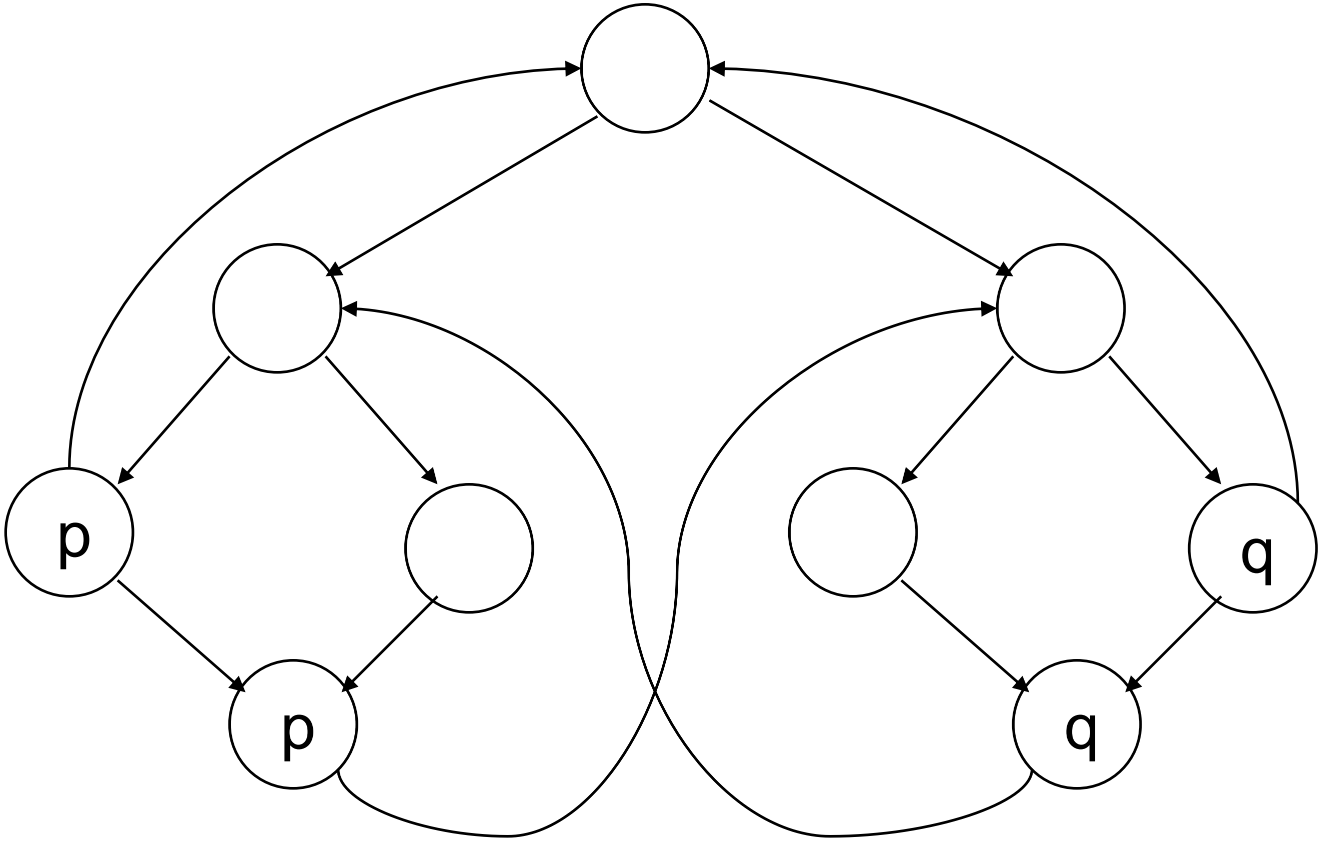
State explosion

- The algorithm is linear in the size of the model but the size of the model is exponential in the number of variables, components, etc.

Can we reduce state explosion?

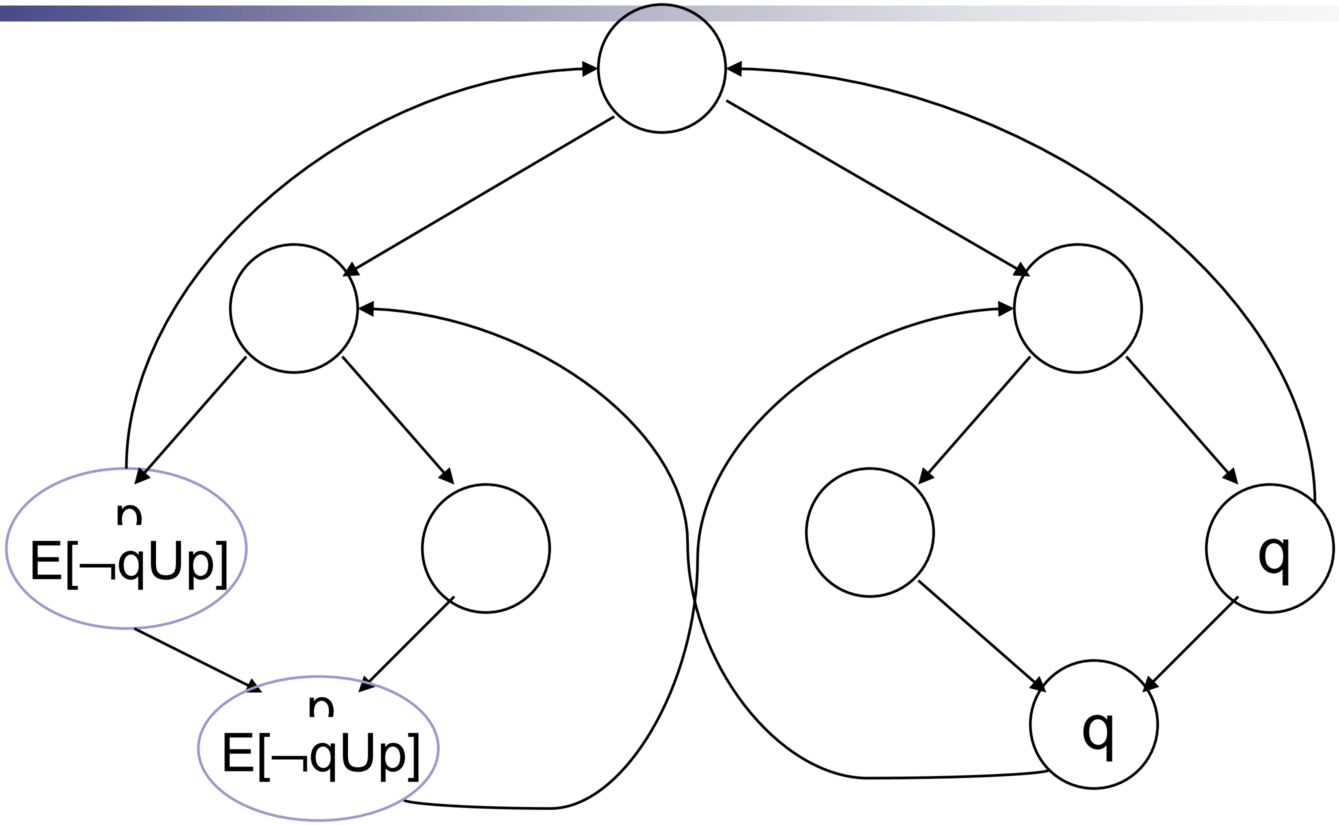
  - Abstraction (what is relevant?)
  - Induction (for ‘similar’ components)
  - Composition (divide and conquer)
  - Reduction (prove semantic equivalence)
  - Ordered binary decision diagrams

Example: Input



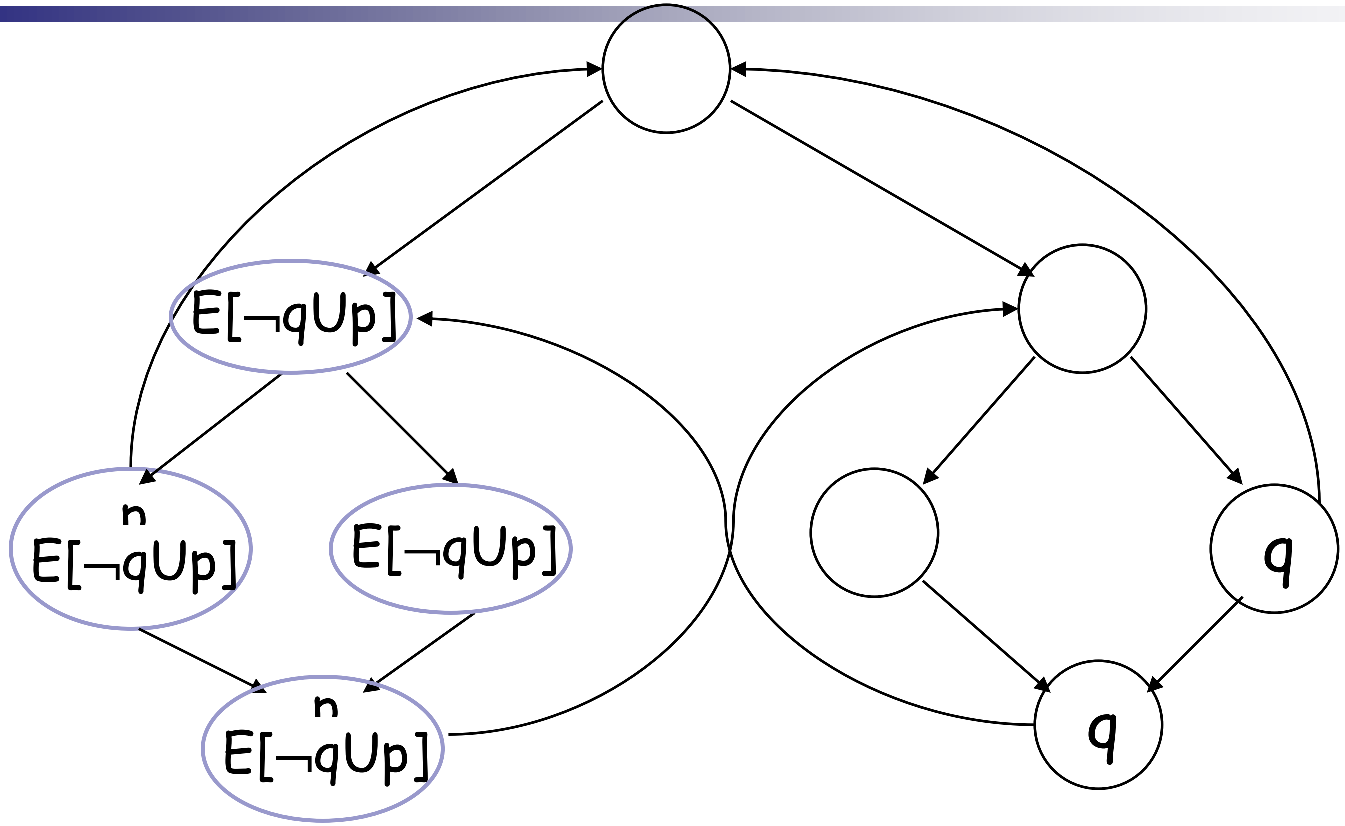
ϕ = AF(E[¬q U p] v EXq)

Example: EU - step 1



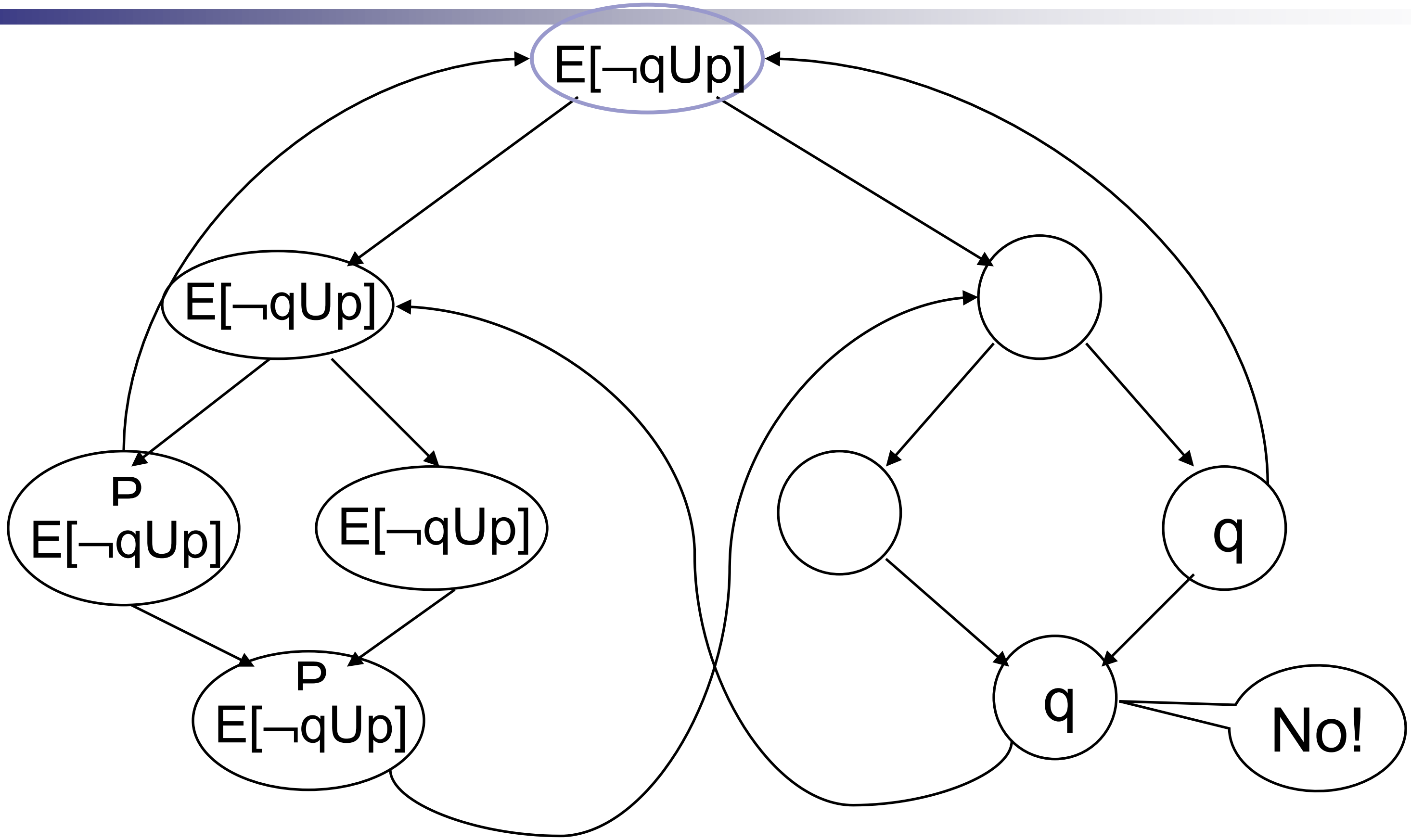
1. Label with E[¬qUp] all states which satisfy p

Example: EU-step 2.1



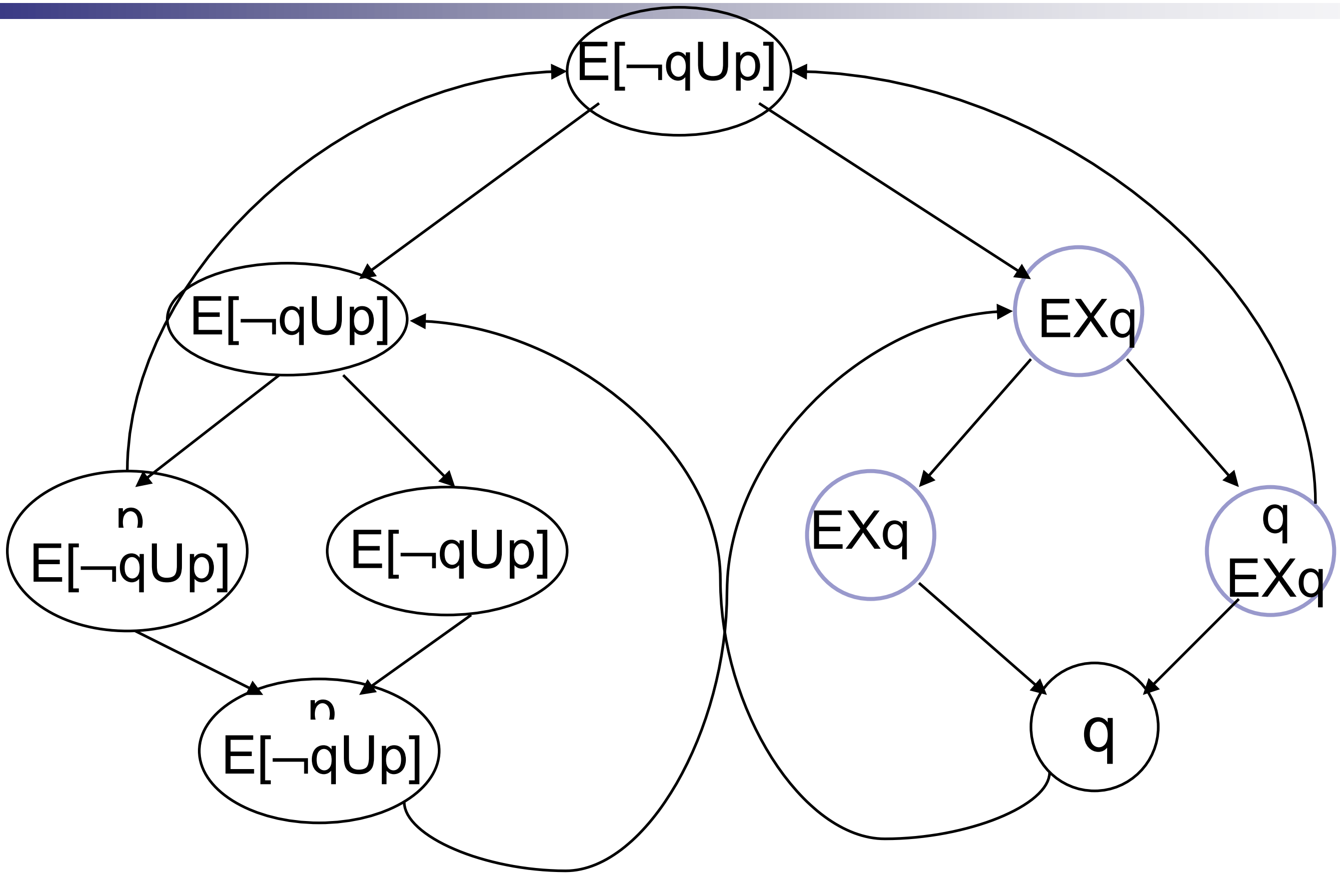
2.1 label with E[¬qUp] any state that is already labeled with ¬q and with one of its successor already labeled by E[¬qUp]

Example: EU-step 2.2



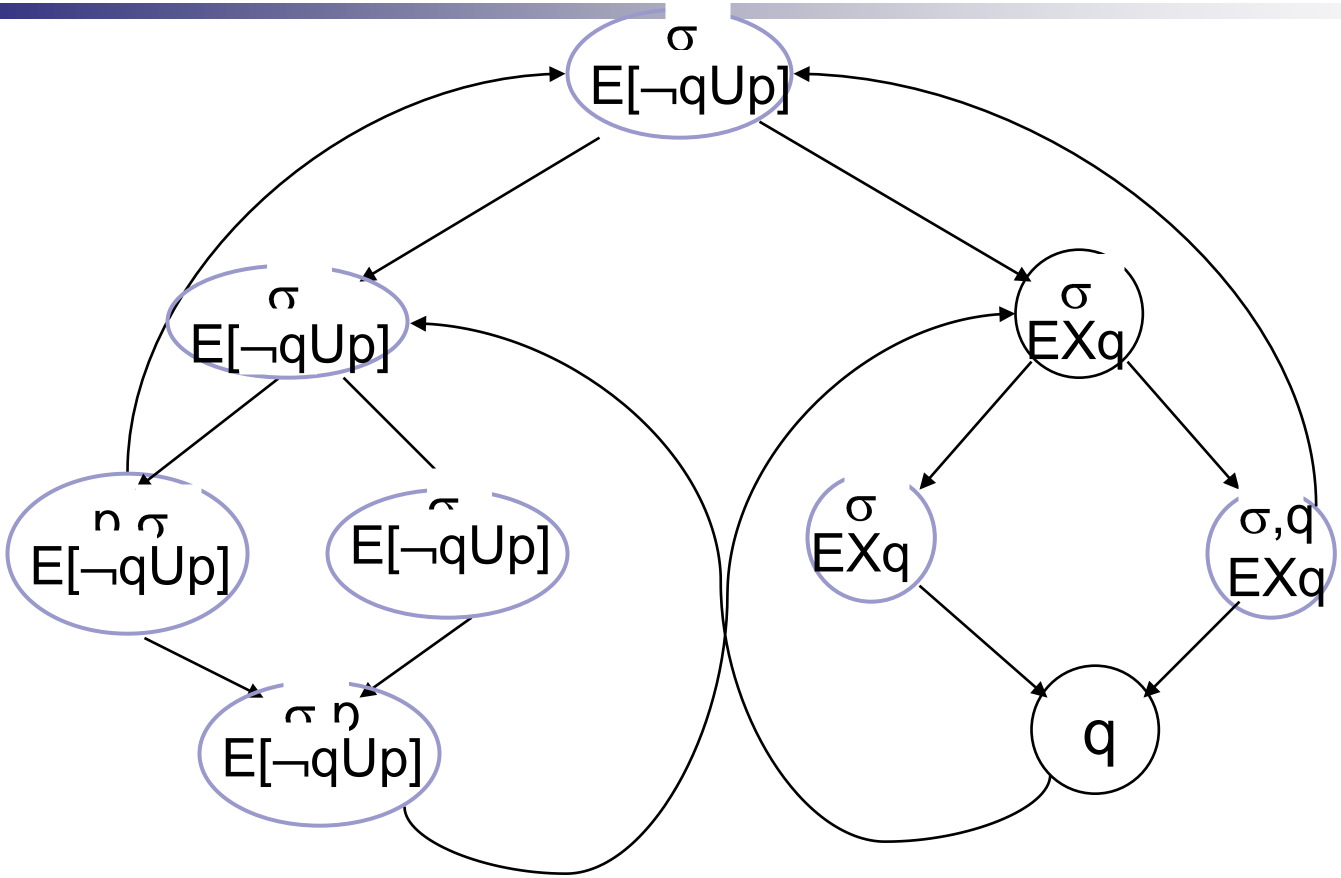
2.2 label with E[¬qUp] any state that is already labeled with ¬q and with one of its successor already labeled by E[¬qUp]

Example: EX-step 3



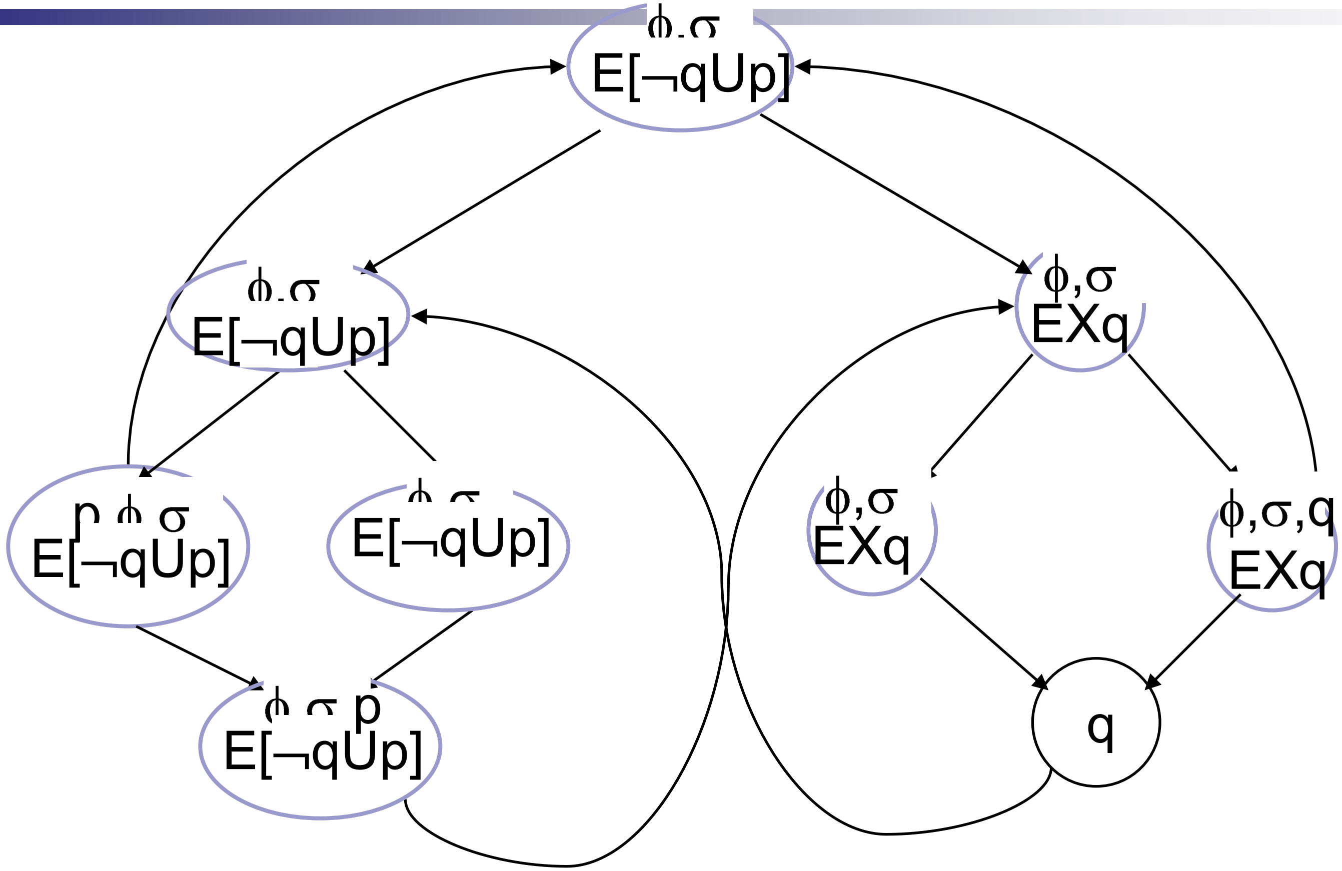
3. Label with EXq any state with one of it successors already labeled by q

Example: ∨-step 4



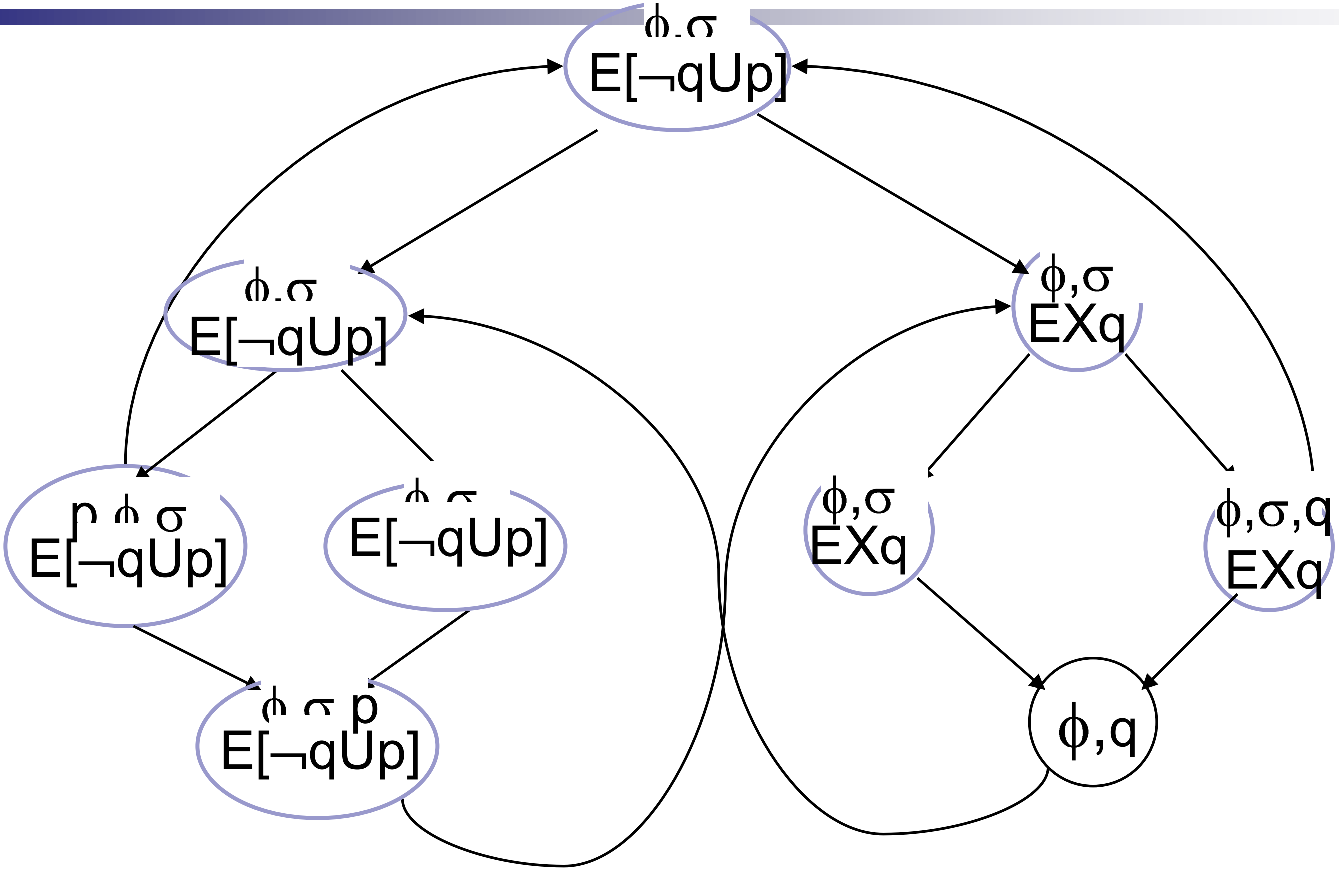
4. Label with σ = E[¬qUp] v EXq any state s already labeled by E[¬qUp] or EXq

Example: AF-step 5.1



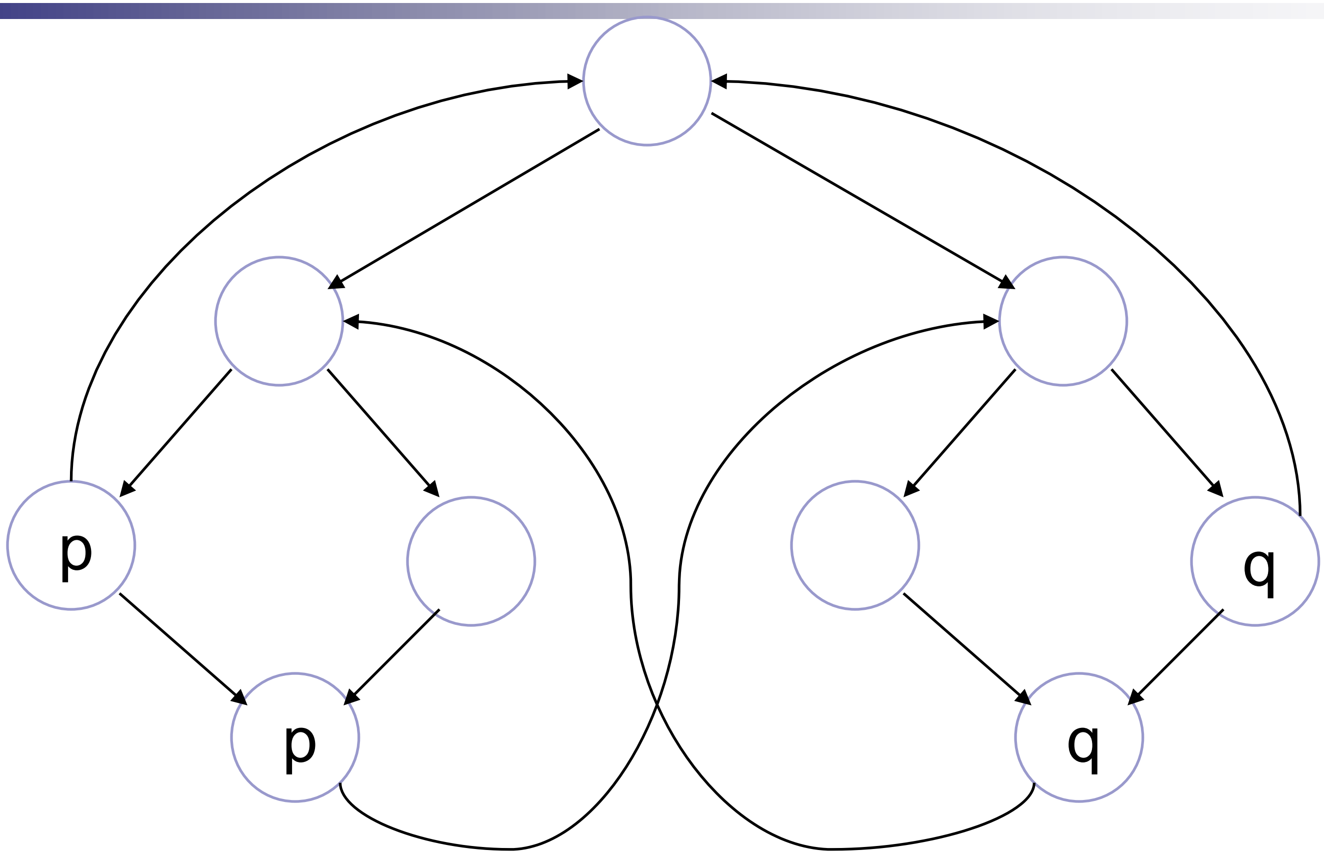
5.1 Label with ϕ = AF(E[¬qUp]vEXq) any state already labeled by σ = E[¬qUp]vEXq

Example: AF-step 5.2



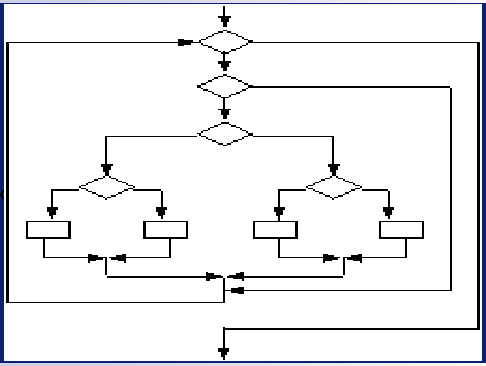
5.2 Label with ϕ any state with all successor already labeled by ϕ.

Example: Output



- All states satisfy AF(E[¬q U p] v EXq)

Spring 2007



Program correctness

SAT and its correctness

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

Context

1. We have defined the semantics of CTL formulas $M,s \models \phi$
2. We have given an efficient method for model checking a CTL formula returning all states s such that $M,s \models \phi$

Next we present an algorithm for it and proves its correctness

The algorithm SAT

- SAT stands for ‘satisfies’
  - Input: a well-formed CTL formula
  - Output: a subset of the states of a transition system M = <S, →,I>

- Written in Pascal-like
  - function return
  - local_var
  - while do od
  - case is end_case

The main function (I)

```
function SAT($\phi$)
begin
    case $\phi$ is
        T : return S
        $\bot$ : return $\varnothing$
        atomic : return $\{s \in S \mid \phi \in l(s)\}$
        $\neg\phi_1$ : return $S - SAT(\phi_1)$
        $\phi_1 \wedge \phi_2$ : return $SAT(\phi_1) \cap SAT(\phi_2)$
        $\phi_1 \vee \phi_2$ : return $SAT(\phi_1) \cup SAT(\phi_2)$
        $\phi_1 \Rightarrow \phi_2$ : return $SAT(\neg\phi_1 \vee \phi_2)$
        ⋮
```

The main function (II)

```
        ⋮
        $AX\phi_1$ : return $SAT(\neg EX\neg\phi_1)$
        $EX\phi_1$ : return $SAT\_EX(\phi_1)$
        $A[\phi_1 U \phi_2]$ : return
                   $SAT(\neg E[\neg\phi_2 U(\neg\phi_1\wedge\neg \phi_2)]\vee EG\neg\phi_2)$
        $E[\phi_1 U \phi_2]$ : return $SAT\_EU(\phi_1, \phi_2)$
        $EF\phi_1$ : return $SAT(E[T U \phi_1])$
        $AF\phi_1$ : return $SAT\_AF(\phi_1)$
        $EG\phi_1$ : return $SAT(\neg AF\neg\phi_1)$          /*$SAT\_EG(\phi_1)$*/
        $AG\phi_1$ : return $SAT(\neg EF\neg\phi_1)$
    end_case
end
```

The function SAT_EX

```
function SAT_EX($\phi$)
local_var X,Y
begin
    X := SAT($\phi$)
    Y := $\{ s \in S \mid \exists s \rightarrow s' : s' \in X\}$
    return Y
end
```

The function SAT_AF

```
function SAT_AF($\phi$)
local_var X,Y
begin
    X := S
    Y := SAT($\phi$)
    while $X \neq Y$ do
        X := Y
        $Y := Y \cup \{ s \in S \mid \forall s \rightarrow s' : s' \in Y \}$
    od
    return Y
end
```

The function SAT_EU

```
function SAT_EU($\phi$,$\psi$)
local_var W,X,Y
begin
    W := SAT($\phi$)
    X := S
    Y := SAT($\psi$)            /* Calculated only once  */
    while X $\neq$ Y do
        X := Y
        Y := Y $\cup$ (W $\cap$ { s $\in$ S | $\exists$s $\to$ s' : s' $\in$ Y })
    od
    return Y
end
```

The function SAT_EG

```
function SAT_EG($\phi$)
local_var X,Y
begin
    X := $\emptyset$
    Y := SAT($\phi$)
    while X $\neq$ Y do
        X := Y
        Y := Y $\cap$ { s $\in$ S | $\exists$s $\rightarrow$ s' : s' $\in$ Y }
    od
    return Y
end
```

Does it work?

- Claim: For a given model $M=<S, \rightarrow, I>$ and well-formed CTL formula ϕ,

$$SAT(\phi) = \{ s \in S \mid M,s \vDash \phi\} \overset{def}{=} [[\phi]]$$

Is this true?

The proof (I)

- The claim is proved by induction on the structure of the formula.
- For ϕ = T, $\bot$, or atomic the set $[[\phi]]$ is computed directly
- For $\neg\phi$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ or $\phi_1 \Rightarrow \phi_2$ we apply induction and predicate logic equivalences
  - ☐ Example:

$$\begin{aligned} SAT(\phi_1 \vee \phi_2) &= SAT(\phi_1) \cup SAT(\phi_2) \\ &= [[\phi_1]] \cup [[\phi_2]] \qquad \text{(induction)} \\ &= [[\phi_1 \vee \phi_2]] \end{aligned}$$

The proof (II)

- For $EX\phi$ we apply induction

$SAT(EX\phi) = SAT\_EX(\phi)$

$= \{ s \in S \mid \exists\, s \rightarrow s' : s' \in SAT(\phi)\}$

$= \{ s \in S \mid \exists s \rightarrow s' : s' \in [[\phi]]\}$ (induction)

$= \{ s \in S \mid \exists s \rightarrow s' : M,s' \models \phi\}$ (definition [[-]])

$= \{ s \in S \mid M,s \models EX\phi\}$ (definition $\models$)

$= [[EX\phi]]$ (definition [[-]])

The proof (III)

- For $AX\phi$, $A[\phi_1 \; U \; \phi_2]$, $EF\phi$, or $AG\phi$ we can rely on logical equivalences and on the correctness of SAT_EX, SAT_AF, SAT_EU, and SAT_EG
  - Example:

    $SAT(AX\phi) = SAT(\neg EX \neg \phi)$

    $= S - SAT\_EX(\neg\phi)$ (def. $SAT(\neg\phi)$)

    $= S - [[EX\neg\phi]]$ (correctness SAT_EX)

    $= [[AX\phi]]$ (logical equivalence)

But we still have to prove the correctness of SAT_AF, SAT_EU, and SAT_EG

EG as fixed point

Recall that $EG\phi \equiv \phi \wedge EX\, EG\phi$. Since

$$EX\psi = \{ s \in S \mid \exists\, s \rightarrow s' : s' \in [[\psi]]\}$$

we have the following fixed-point definition of EG

$$[[EG\phi]] = [[\phi]] \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in [[EG\phi]]\}$$

Fixed points

- Let S be a set and F:Pow(S) → Pow(S) be a a function
  - ☐ F is monotone if

    $$X \subseteq Y \text{ implies } F(X) \subseteq F(Y)$$

    for all subsets X and Y of S

  - ☐ A subset X of S is a fixed point of F if

    $$F(X) = X$$

  - ☐ A subset X of S is a least fixed point of F if

    $$F(X) = X \text{ and } X \subseteq Y$$

    for all fixed point Y of F

Examples

- S = {s,t} and $F: X \mapsto X \cup \{s\}$
  - ☐ F is monotone
  - ☐ {s} and {s,t} are all fixed points of F
  - ☐ {s} is the least fixed point of F

- S = {s,t} and G:X↦<u>if</u> X={s} <u>then</u> {t} <u>else</u> {s}
  - ☐ G is not monotone
    - $\{s\} \subseteq \{s,t\}$ but $G(\{s\}) = \{t\} \not\subset \{s\} = G(\{s,t\})$
  - ☐ G does not have any fixed point

Fixed points (II)

Let $F^i(X) = \underbrace{F(F(\ldots F}_{\text{i-times}}(X)\ldots))$ for $i > 0$ (thus $F^1(X) = F(X)$)

- Theorem: Let S be a set with n+1 elements. If $F : Pow(S) \rightarrow Pow(S)$ is a monotone function then
  1) $F^{n+1}(\emptyset)$ is the least fixed point of F
  2) $F^{n+1}(S)$ is the greatest fixed point of F

Least and greatest fixed points can be computed and the computation is guaranteed to terminate ! ☺

Computing EGϕ

- To find a set $[[EG\phi]]$ such that

  $$[[EG\phi]] = [[\phi]] \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in [[EG\phi]]\}$$

  we look if it is a fixed point of the function

  $$F(X) = [[\phi]] \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in X\}$$

- **Theorem:** Let $n = |S|$ be the size of S and F defined as above. We have
  1. F is monotone
  2. $[[EG\phi]]$ is the greatest fixed point of F
  3. $[[EG\phi]] = F^{n+1}(S)$

Correctness of SAT_EG

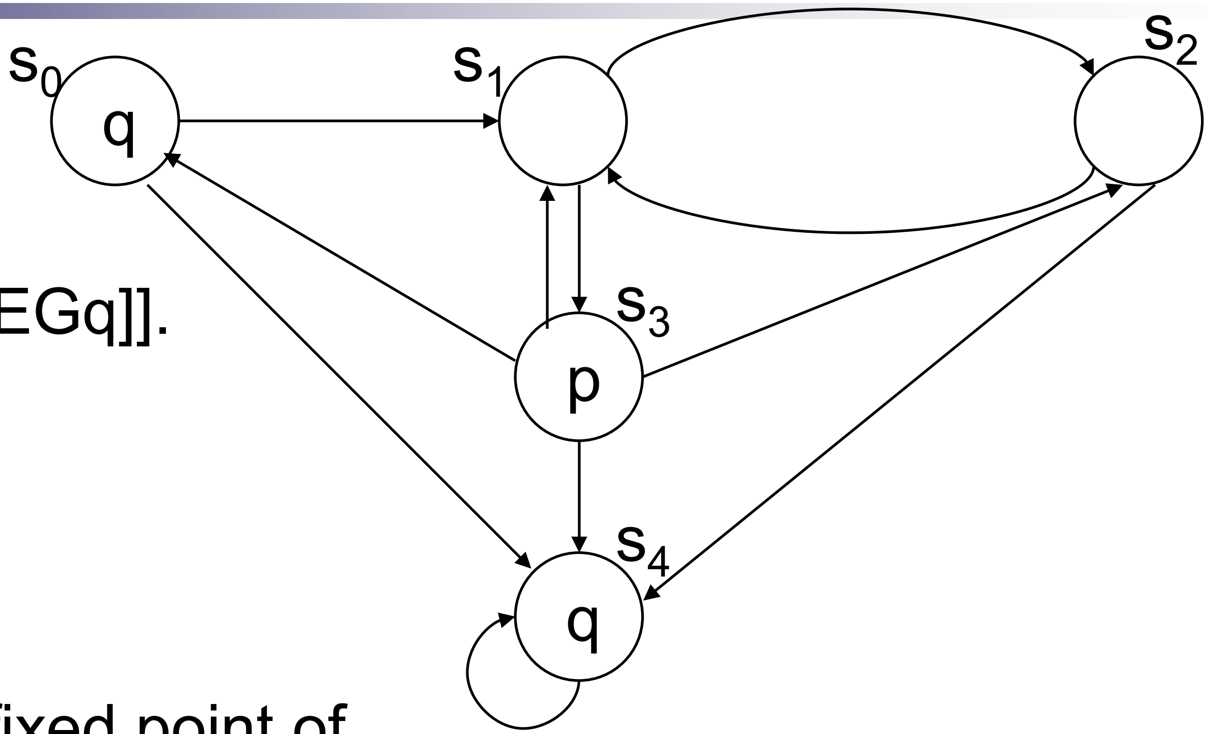
1. Inside the loop it always holds $Y \subseteq SAT(\phi)$
2. Because $Y \subseteq SAT(\phi)$, substitute in SAT_EG

   $Y := Y \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in Y \}$

   with $Y := SAT(\phi) \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in Y \}$
3. Note that $SAT\_EG(\phi)$ is calculating the greatest fixed point (use induction!)

   $$F(X) = [[\phi]] \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in X\}$$
4. It follows from the previous theorem that $SAT\_EG(\phi)$ terminates and computes $[[EG\phi]]$.

Example: EG



Let us compute [[EGq]].

It is the greatest fixed point of

$$F(X) = [[q]] \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in X \}$$

$$= \{s_0, s_4\} \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in X \}$$

Example: EG

- Iterating F on S until it stabilizes
  - ☐ $F^1(S) = \{s_0,s_4\} \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in S \}$

    $= \{s_0,s_4\} \cap S$

    $= \{s_0,s_4\}$

  - ☐ $F^2(S) = F(F^1(S))$

    $= F(\{s_0,s_4\})$

    $= \{s_0,s_4\} \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in \{s_0,s_4\} \}$

    $= \{s_0,s_4\}$

- Thus $\{s_0,s_4\}$ is the greatest fixed point of F and equals [[EGq]]

EU as fixed point

- Recall that $E[\phi \text{ U } \psi] \equiv \psi \vee (\phi \wedge EX\, E[\phi \text{ U } \psi])$.
- Since $EX\varphi = \{ s \in S \mid \exists\, s \rightarrow s' : s' \in [[\varphi]]\}$ we obtain

$$[[E[\phi U \psi]]] = [[\psi]] \cup ([[\phi]] \cap \{s \in S \mid \exists s \rightarrow s' : s' \in [[E[\phi U \psi]]]\})$$

Computing $E[\phi \text{ U } \psi]$

- As before, we show that $[[E[\phi \text{ U } \psi]]]$ is a fixed point of the function

  $$G(X) = [[\psi]] \cup ([[\phi]] \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in X\})$$

- Theorem: Let n = |S| be the size of S and G defined as above. We have
  1. G is monotone
  2. $[[E[\phi \text{ U } \psi]]]$ is the least fixed point of G
  3. $[[E[\phi \text{ U } \psi]]] = G^{n+1}(\emptyset)$

Correctness of SAT_EU

1. Inside the loop it always holds $W=SAT(\phi)$ and $Y \supseteq SAT(\psi)$.

2. Substitute in SAT_EU

   $$Y:=Y \cup (W \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in Y \})$$

   with

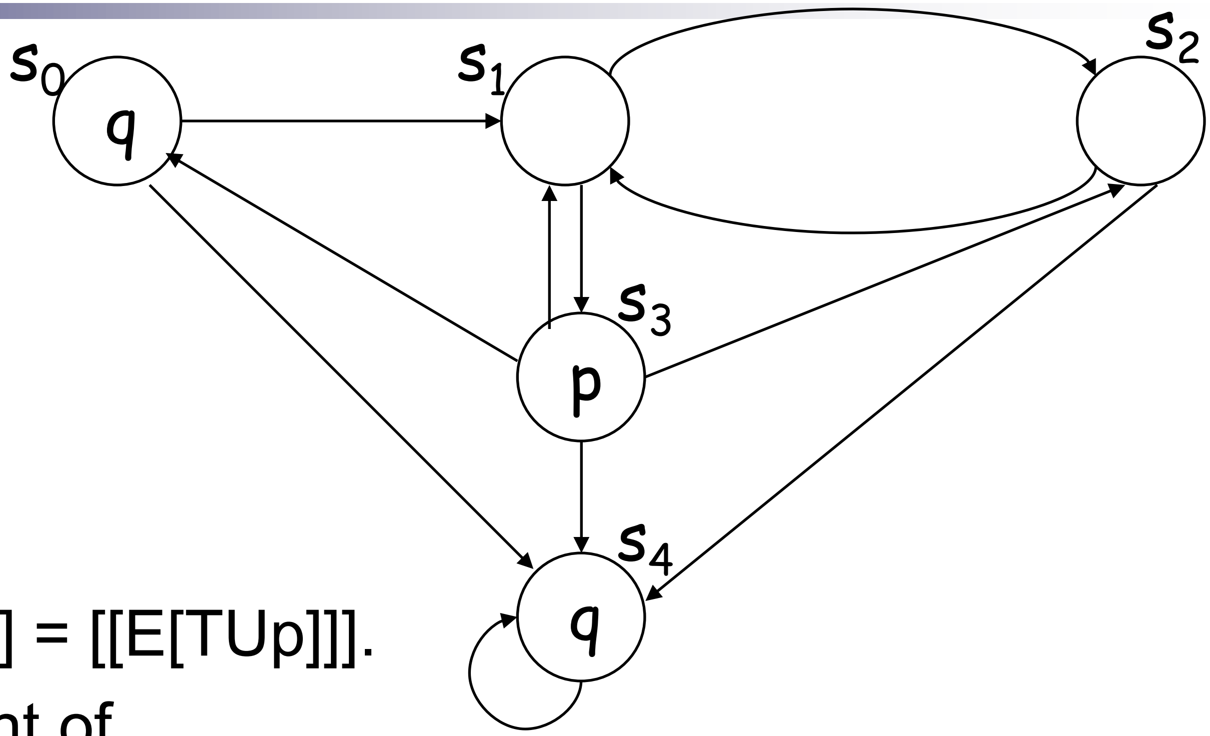
   $$Y:=SAT(\psi) \cup (SAT(\phi) \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in Y \})$$

3. Note that SAT_EU(ϕ) is calculating the least fixed point of

   $$G(X) = [[\psi]] \cup ([[\phi]] \cap \{ s \in S \mid \exists s \rightarrow s' : s' \in X\})$$

4. It follows from the previous theorem that SAT_EU(ϕ,ψ) terminates and computes $[[E[\phi U \psi]]]$

Example: EU



Let us compute [[EFp]] = [[E[TUp]]].

It is the least fixed point of

$$G(X) = [[p]] \cup ([[T]] \cap \{ s \in S \mid \exists s \to s' : s' \in X\})$$
$$= \{s_3\} \cup (S \cap \{ s \in S \mid \exists s \to s' : s' \in X \})$$
$$= \{s_3\} \cup \{ s \in S \mid \exists s \to s' : s' \in X \}$$

Example: EU

- Iterating G on $\emptyset$ until it stabilizes we have
  - $G^1(\emptyset) = \{s_3\} \cup \{ s \in S \mid \exists s \to s' : s' \in \emptyset \}$
    $= \{s_3\} \cup \emptyset = \{s_3\}$
  - $G^2(\emptyset) = G(G^1(\emptyset)) = G(\{s_3\})$
    $= \{s_3\} \cup \{ s \in S \mid \exists s \to s' : s' \in \{s_3\} \}$
    $= \{s_1,s_3\}$
  - $G^3(\emptyset) = G(G^2(\emptyset)) = G(\{s_1,s_3\})$
    $= \{s_3\} \cup \{ s \in S \mid \exists s \to s' : s' \in \{s_1,s_3\} \}$
    $= \{s_0,s_1, s_2,s_3\}$
  - $G^4(\emptyset) = G(G^3(\emptyset)) = G(\{s_0,s_1, s_2,s_3\})$
    $= \{s_3\} \cup \{ s \in S \mid \exists s \to s' : s' \in \{s_0,s_1, s_2,s_3\} \}$
    $= \{s_0,s_1, s_2,s_3\}$
- Thus [[EFp]] = [[E[TUp]]] $= \{s_0,s_1,s_2,s_3\}$.

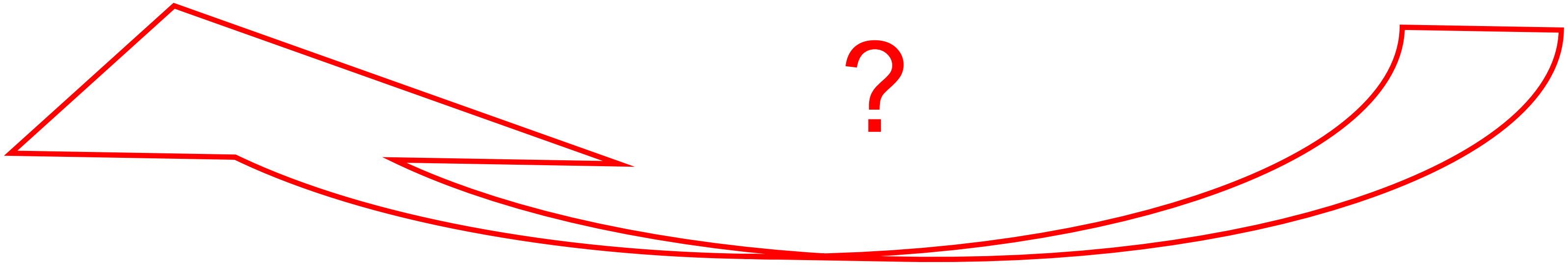
AF as fixed point

Since $\mathrm{AF}\phi \equiv \phi \vee \mathrm{AX}\ \mathrm{AF}\phi$ and

$$\mathrm{AX}\varphi = \{\, s \in S \mid \forall s \to s' : s' \in [[\varphi]]\}$$

we obtain

$$[[\mathrm{AF}\phi]] = [[\phi]] \cup \{\, s \in S \mid \forall s \to s' : s' \in [[\mathrm{AF}\phi]]\}$$



Computing AFϕ

- Again, consider $[[AF\phi]]$ as a fixed point of the function

$$H(X) = [[\phi]] \cup \{ s \in S \mid \forall s \to s' : s' \in X\}$$

- Theorem: Let n = |S| be the size of S and G defined as above. We have
  1. H is monotone
  2. $[[AF\phi]]$ is the least fixed point of H
  3. $[[AF\phi]] = H^{n+1}(\emptyset)$

Correctness of SAT_AF

1. Inside the loop it always holds $Y \supseteq SAT(\phi)$.
2. Substitute in SAT_AF

   $$Y{:=}Y \cup \{\, s \in S \mid \forall s \to s' : s' \in Y \,\})$$

   with

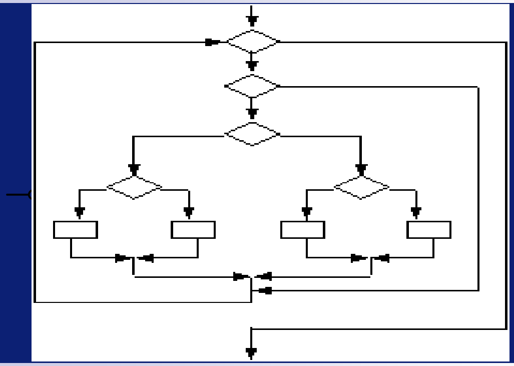
   $$Y{:=}SAT(\phi) \cup \{\, s \in S \mid \forall s \to s' : s' \in Y \,\}$$

3. Note that SAT_AF(ϕ) is calculating the least fixed point of

   $$H(X) = [[\phi]] \cup \{\, s \in S \mid \forall s \to s' : s' \in X\}$$

4. It follows from the previous theorem that AT_AF(ϕ) terminates and computes [[**AF**ϕ]]

Spring 2007



Program correctness

Model checking LTL

Marcello Bonsangue

Context

- Model checking CTL was relatively easy because the truth of formulas depends

  on the current state (CTL)

  and not

  on an execution path (LTL)

  and not

  on the tree of all executions (CTL*)

- Next we concentrate on model checking LTL

LTL: a recap

- Syntax

  $\phi ::= \top \mid p \mid \neg\phi \mid \phi\vee\phi \mid X\phi \mid \phi U\phi$

  All other connectives can be written in the above syntax

LTL formulas as languages (I)

- ϕ = GFp (infinitely often p)
  - The execution $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \ldots$ satisfies ϕ if it contains infinitely many s_{n_1}, s_{n_2}, … at which p holds. In between there can be an arbitrary but finite number of state at which $\neg$p holds.

As a language $((\neg p)^*.p)^\omega$

ω-regular expressions

* = an arbitrary but finite number of repetitions

$^\omega$ = an infinite number of repetitions

LTL formulas as languages(II)

- ϕ = FGp (Eventually always p)
- The execution $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \ldots$ satisfies ϕ if from a certain state onwards at all states p holds.
- As ω-regular expression $(p + \neg p)^*.p^\omega$

Automata on finite words: a recap

- A non-deterministic finite automaton is a special kind of transition systems for recognizing languages on finite words

- NF-automaton $A = < \Sigma, S, \rightarrow, I, F>$

Σ	finite alphabet
S	finite set of states
$\rightarrow \subseteq S \times \Sigma \times S$	transition relation
$I \subseteq S$	initial states
$F \subseteq S$	accepting states

- The language of an automaton A is

$$L(A) = \{a_1a_2 \ldots a_n \in \Sigma^* \mid \exists\, s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_3} s_n \in F \text{ with } s_1 \in I\}$$

Properties of finite languages

- Theorem: $L(A_1 x\ A_2) = L(A_1) \cap L(A_2)$

  $A_1 x\ A_2 = <\Sigma, S_1 x S_2, \rightarrow, I_1 x I_2, F_1 x F_2>$ where

  $<s,t> \xrightarrow{a} <s',t'>$ iff $s \xrightarrow{a}_1 s'$ and $t \xrightarrow{a}_2 t'$

- Theorem: $L(A) = \emptyset$ is decidable

  It is enough to find a path from an initial state in I to a final state in F.

Automata on infinite words: Buchi

- A Buchi automaton is a special kind of transition systems for recognizing languages on infinite words

- Buchi automaton $A = <\Sigma, S, \rightarrow, I, F>$

Σ	finite alphabet
S	finite set of states
$\rightarrow \subseteq S \times \Sigma \times S$	transition relation
$I \subseteq S$	initial states
$F \subseteq S$	accepting states

Buchi automata

An infinite execution of a Buchi automaton A

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} s_4 \ldots$$

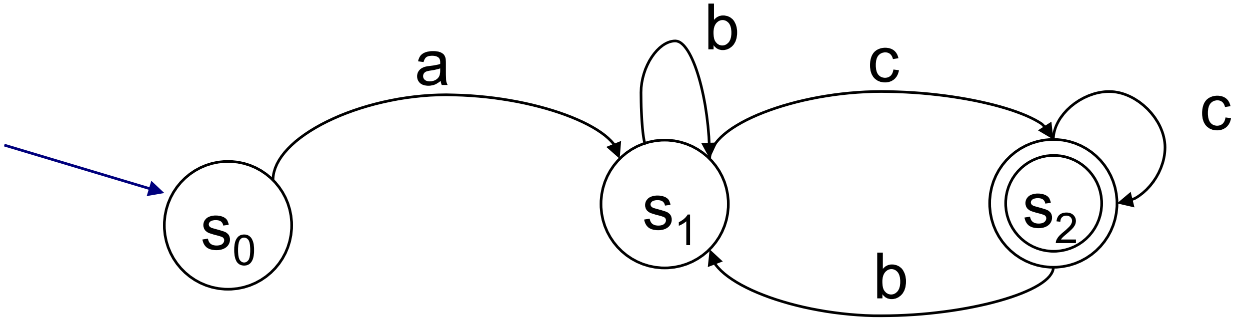
is accepted by A if

- ☐ $s_1 \in I$
- ☐ there exists infinitely many $i > 0$ such that $s_i \in F$

- The language of a Buchi automaton A is

$L_\omega(A) = \{a_1a_2\ldots \in \Sigma^\omega \mid \exists\ s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \text{ accepted by } A\}$

Example



- abccccccc... accepted
- abcbcbcbcb... accepted
- abcbbbbbb… rejected

Properties of infinite languages

- **Theorem**: $L_\omega(A_1 \otimes A_2) = L_\omega(A_1) \cap L_\omega(A_2)$

  $A_1 \otimes A_2 = <\Sigma, S_1 x S_2 x \{1,2\}, \rightarrow, I_1 x I_2 x \{1\}, F_1 x S_2 x \{1\}>$

  where $<s,t,i> \xrightarrow{a} <s',t',j>$ iff

  - $s \xrightarrow{a}_1 s'$ and $t \xrightarrow{a}_2 t'$ and i=j unless
  - i=1 and $s \in F_1$ in which case j = 2, or
  - i=2 and $t \in F_2$ in which case j =1.

- **Theorem**: $L_\omega(A) = \varnothing$ is decidable

  It is enough to find a path from an initial state $s \in I$ to a final state $t \in F$ such that t has a path to t itself.

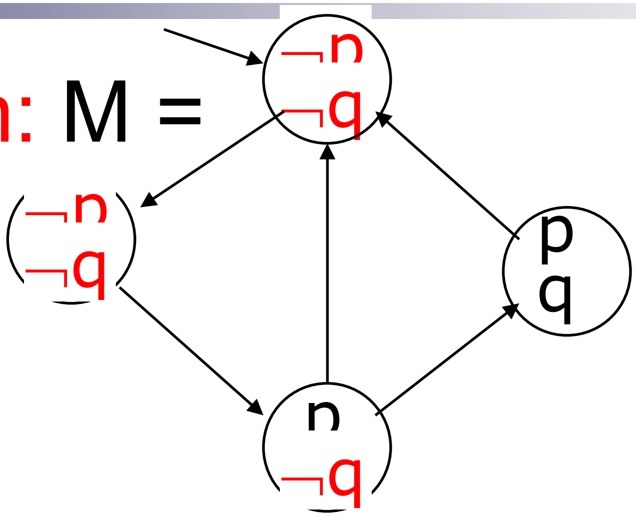
Transition systems and Buchi automata

- Any transition systems $M = \langle S, \rightarrow_M, s_0 \rangle$ with a labelling function $\ell : S \rightarrow 2^{Prop}$ can be seen as a Buchi automata $A_M = \langle \Sigma, S, \rightarrow, I, F \rangle$ where

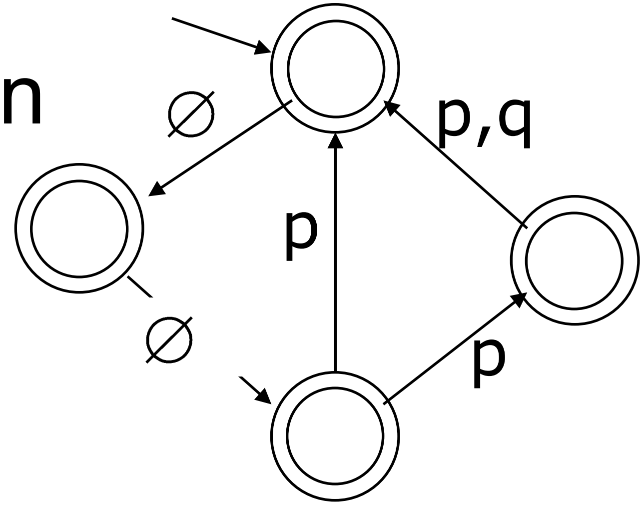
$\Sigma = 2^{Prop}$	assignment of truth values to propositions (i.e. valuations)
S	same states
$s \xrightarrow{a} t$ iff $s \rightarrow_M t$ and $a = \ell(s)$	transition relation
$I = \{s_0\}$	same initial state
$F = S$	every state is final

Example

- The system: M =



becomes the Buchi automaton



LTL and Buchi automata

- An LTL formula denotes a set of infinite traces which satisfy that formula

- A Buchi automaton accepts a set of infinite traces

- Theorem: Given an LTL formula ϕ, we can build a Buchi automaton

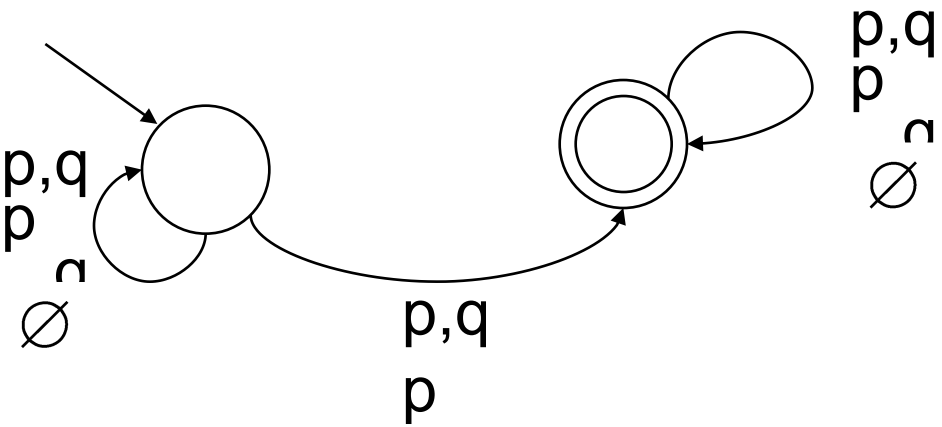
  $$A_{\phi} = < \Sigma, S, \rightarrow, I, F>$$

  where $\Sigma = 2^{Prop}$ consists of the subsets of (possibly negated) atomic propositions (i.e. valuations), which accepts only and all the executions satisfying the formula ϕ.

Example (1)

- $\phi = Fp$ eventually p

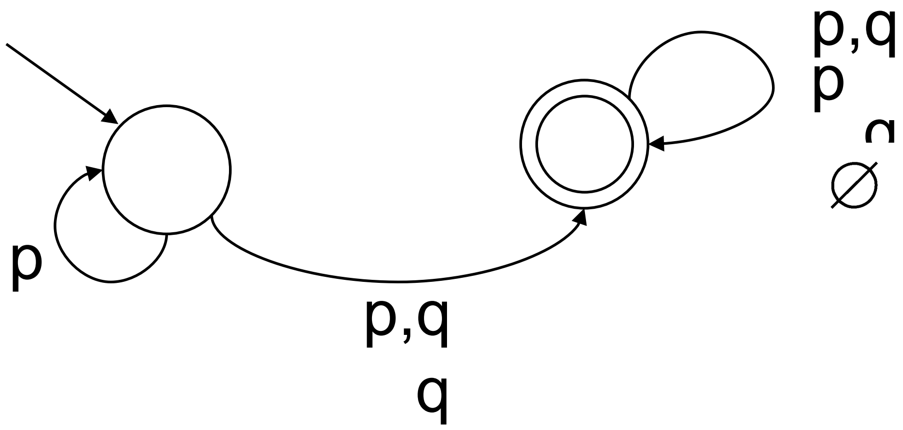
$A_\phi =$



Example (2)

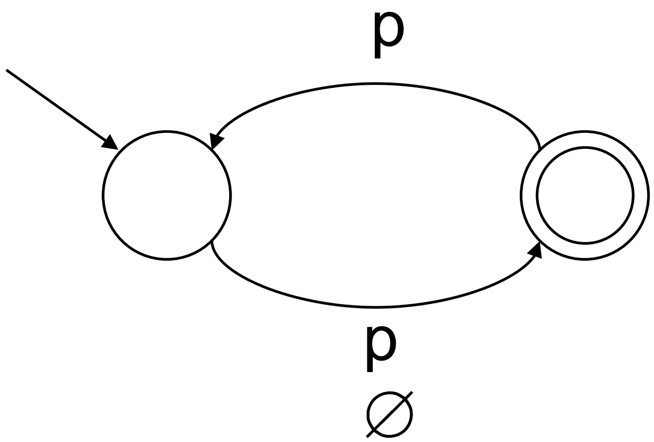
- $\phi = p \, U \, q$ p until q

$A_\phi =$



LTL and Buchi automata

- Not every Buchi automaton is an LTL formula:



“p holds on every odd step”

Model checking LTL:the idea

- Let ϕ be an LTL formula and M,s be a transition system specifying the behavior of a system
  - A_ϕ corresponds to all allowable behavior of the system
  - A_M corresponds to all possible behavior of the system (all infinite paths of M that are potentially interesting)

  To see whether a system satisfies a specification we need to check if every path of A_M is in A_ϕ

$$L_\omega(A_M) \subseteq L_\omega(A_\phi)$$

Model checking LTL

- To check set inclusion note that

$$B \subseteq A \Leftrightarrow B \cap \bar{A} = \emptyset$$

- Further, $\overline{L_\omega(A_\phi)} = L_\omega(A_{\neg\phi})$ thus

Every possible path is allowable
is equivalent to say that
there is no path that is possible and not allowable

that is $M,s \vDash \phi$ if and only if $L_\omega(A_M) \cap L_\omega(A_{\neg\phi}) = \emptyset$

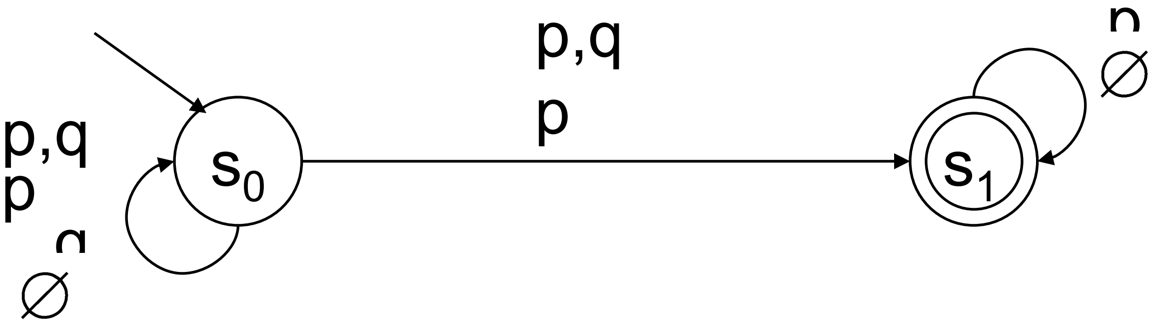
The method

- Problem: $M,s \models \phi$?

1. Construct a Buchi automaton $A_{\neg\phi}$ representing the negation of the desired LTL specification ϕ
2. Construct the automaton A_M representing the system behavior
3. Construct the automaton $A_M \otimes A_{\neg\phi}$
4. Check if $L_\omega(A_M \otimes A_{\neg\phi}) = \varnothing$
5. If yes then $M,s \models \phi$

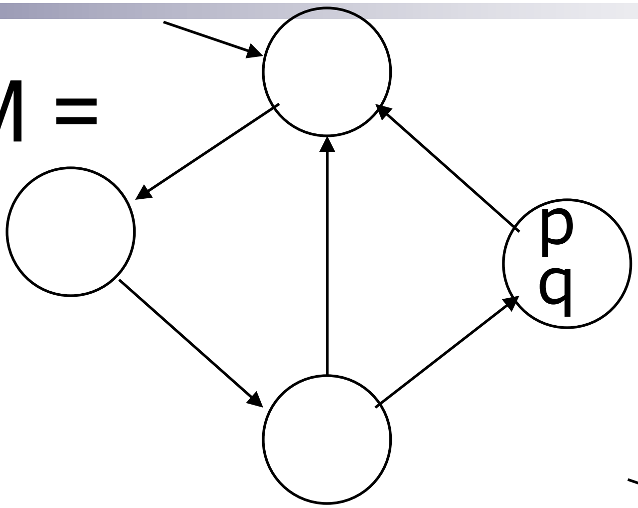
Example (1)

- Specification: $\phi = G(p \Rightarrow XFq)$
  Any occurence of p must be followed (later) by an occurrence of q
- $\neg\phi = F(p \wedge XG\neg q)$
  there exist an occurrence of p after which q will never be encountered again
- $A_{\neg\phi}$ =

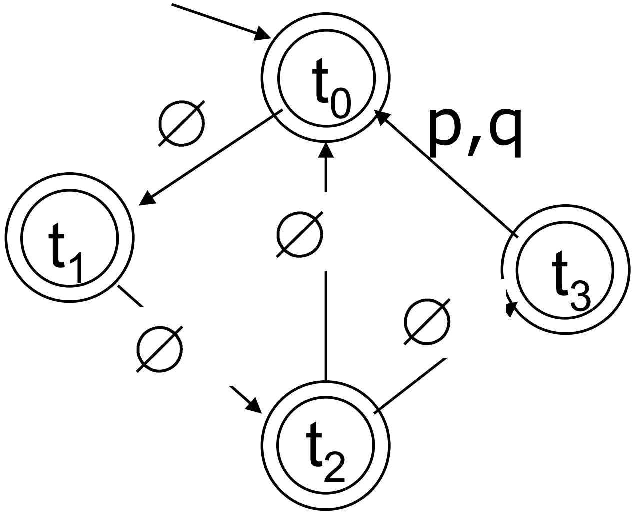


Example (2)

- The system: M =

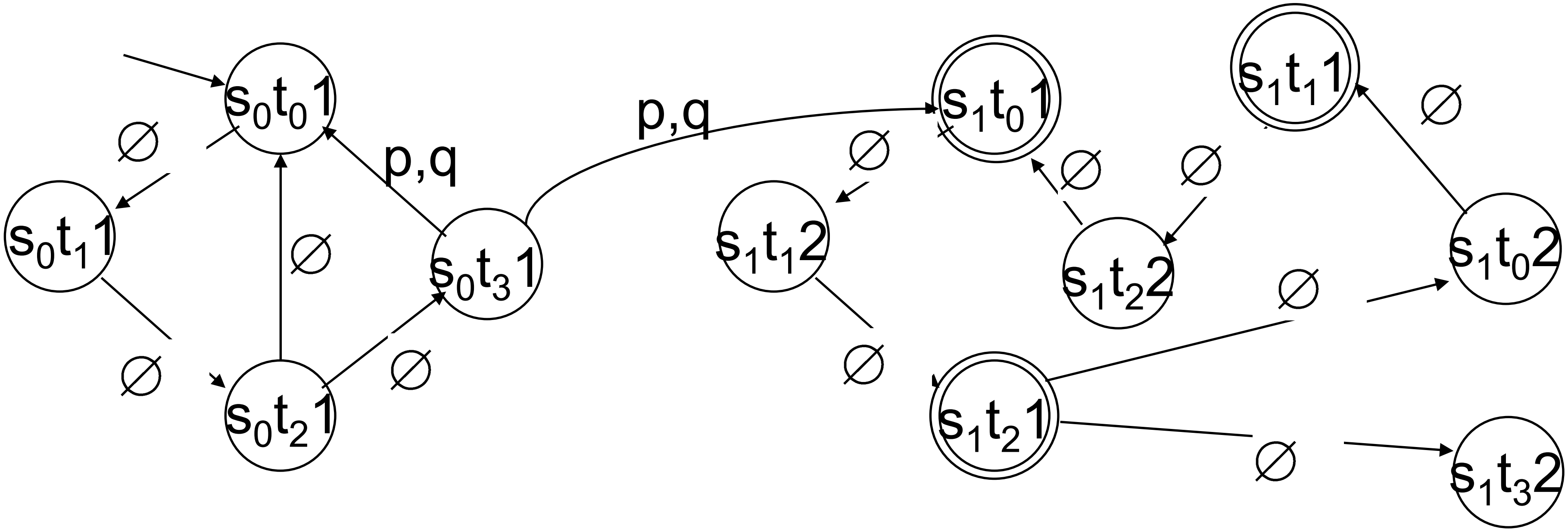


and its Buchi automaton A_M



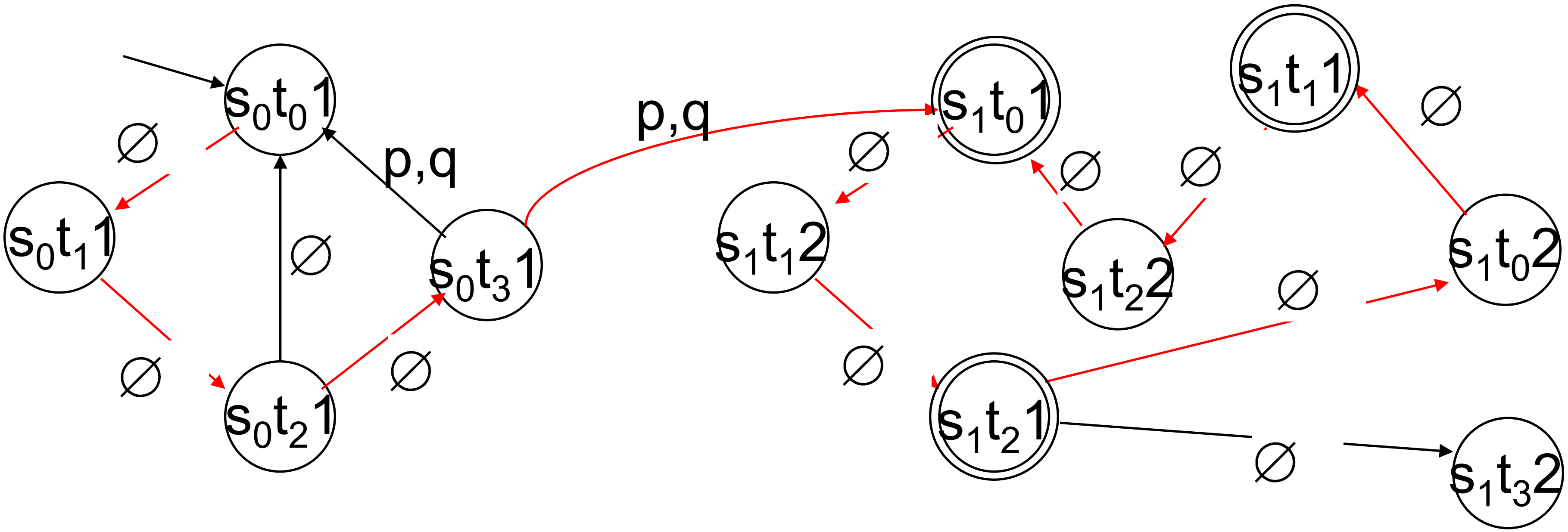
Example: (3)

- The product $A_{\neg\phi} \otimes A_M$



Example: (4)

- $L(A_{\neg\phi} \otimes A_M) = \emptyset$?



There is a path starting from $<s_0t_01>$ that passes infinitely often through the final states

Example: (5)

- Since $L(A_{\neg\phi} \otimes A_M)$ is not empty

$$M,s \not\models G(p \Rightarrow XFq)$$

The counterexample is given by the path

$$t_0 t_1 t_2 t_3 t_0 t_1 t_2 t_0 t_1 t_2 t_0 \ldots$$

From LTL to Buchi automata

- General approach:
  - Rewrite formula in normal form
  - Translate formula into generalized Buchi automata
  - Turn generalized Buchi automata into ordinary Buchi automata

Normal form

- LTL formulas with the until operator U that may contains also the next operators X

- Every formula ϕ can be converted into an equivalent formula ψ in normal form expressing an infinite behavior using equivalences such as:
  - $T = T \; U \; T$
  - $p = p \wedge XT$
  - $F\phi = T \; U \; \phi$ $\qquad$ $G \phi = \bot \; R \; \phi$
  - $\phi_1 R \phi_2 = \neg(\neg\phi_1 U \neg\phi_2)$

Additional simplifications

- Use extra equivalences to reduce size of the formula. For example:
  - $\neg\neg\phi = \phi$
  - $X\phi_1 \vee X\phi_2 = X(\phi_1 \vee \phi_2)$
  - $X\phi_1 \wedge X\phi_2 = X(\phi_1 \wedge \phi_2)$
  - $X\phi_1 U X\phi_2 = X(\phi_1 U \phi_2)$

Example:

- $G(Fp \Rightarrow q) = G(\neg Fp \vee q)$
  $= \bot\ R\ (\neg Fp \vee q)$
  $= \neg\ (\neg\ \bot\ U\ \neg(\neg\ (T\ U\ p) \vee q))$

- $p \wedge \neg q = (p \wedge \neg q) \wedge T$
  $= (p \wedge \neg q) \wedge XT$
  $= (p \wedge \neg q) \wedge XGT$
  $= (p \wedge \neg q) \wedge X(T\ U\ T)$

Generalized Buchi Automata

- They differ from (normal) Buchi automata only in the acceptance condition, which is a 'set of acceptance sets', i.e. $\mathcal{F} \subseteq 2^S$

- The language of a generalized Buchi automaton

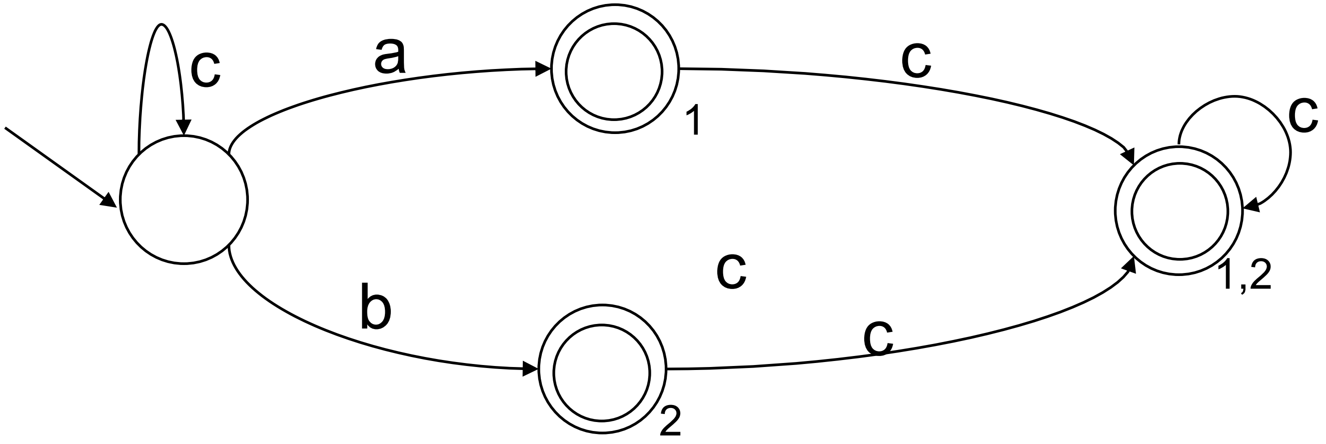
  $A = < \Sigma, S, \rightarrow, I, \mathcal{F} >$ is

  $$L(A) = \cap \{ L(A_F) \mid F \in \mathcal{F} \text{ and } A_F = < \Sigma, S, \rightarrow, I, F> \}$$

  that is, a path has to visit for each set of final states $F \in \mathcal{F}$ infinitely many times states from *F*.

Example

- A generalized Buchi automaton:



- Every path of c’s with either eventually one a or eventually one b is accepted

Generalized Buchi Automata

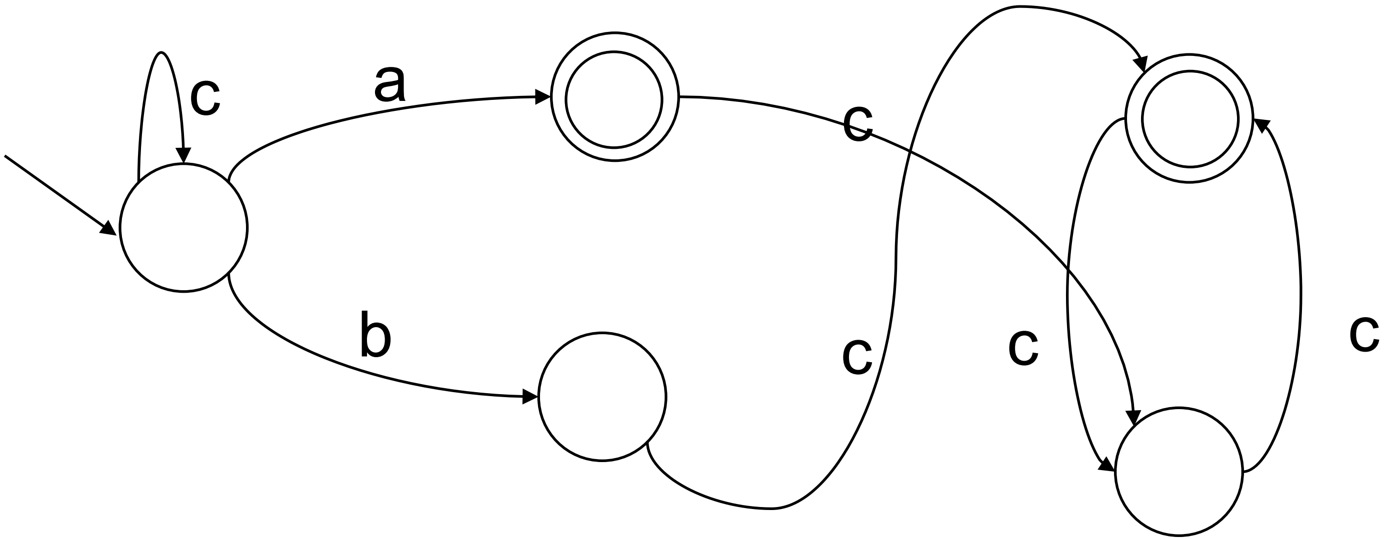
- A generalised Buchi automaton $A = < \Sigma, S, \rightarrow, I, \mathcal{F} >$ can be translated back into an ordinary Buchi automata by taking the intersection of the automata $A_F = < \Sigma, S, \rightarrow, I, F>$ for each $F \in \mathcal{F}$.

- If $\mathcal{F} = \emptyset$ then every infinite path is accepted.
- The ordinary Buchi automata of $< \Sigma, S, \rightarrow, I, \emptyset>$ is

$$< \Sigma, S, \rightarrow, I, S >$$

Example (cont'd)

- The translation of the previous automaton into an ordinary Buchi automaton is



Closure of a formula

- Given an LTL formula ϕ define its closure $\mathrm{Cl}(\phi)$ to be the set of subformulas ψ of ϕ and of their complement.
  - $\phi \in \mathrm{Cl}(\phi)$
  - $\psi \in \mathrm{Cl}(\phi)$ implies $\neg\psi \in \mathrm{Cl}(\phi)$
  - $\psi_1 \vee \psi_2 \in \mathrm{Cl}(\phi)$ implies $\psi_1, \psi_2 \in \mathrm{Cl}(\phi)$
  - $X\psi \in \mathrm{Cl}(\phi)$ implies $\psi \in \mathrm{Cl}(\phi)$
  - $\psi_1 U \psi_2 \in \mathrm{Cl}(\phi)$ implies $\psi_1, \psi_2 \in \mathrm{Cl}(\phi)$

Constructing the automata A_ϕ:states

- The states Sub(ϕ) of the automata are the maximal subsets S of Cl(ϕ) that have no propositional inconsitency

1. For all $\psi \in Cl(\phi)$, $\psi \in S$ iff $\neg\psi \notin S$
2. If $T \in Cl(\phi)$ then $T \in S$
3. $\psi_1 \vee \psi_2 \in S$ iff $\psi_1 \in S$ or $\psi_2 \in S$, whenever $\psi_1 \vee \psi_2 \in Cl(\phi)$
4. $\neg(\psi_1 \vee \psi_2) \in S$ iff $\neg\psi_1 \in S$ and $\neg\psi_2 \in S$, whenever $\neg(\psi_1 \vee \psi_2) \in Cl(\phi)$
5. If $\psi_1 U \psi_2 \in S$ then $\psi_1 \in S$ or $\psi_2 \in S$
6. If $\neg(\psi_1 U \psi_2) \in S$ then $\neg\psi_2 \in S$

Intuition: $\psi \in S$ implies that ψ holds in S

- The initial states are those states containing ϕ

Example

- $Cl(pUq) = \{p,q,\neg p,\neg q, pUq, \neg(pUq)\}$

- $Sub(pUq) = \{\{p, q,pUq\},$
  $\{p,\neg q,pUq\},$
  $\{p,\neg q,\neg(pUq)\}$
  $\{\neg p,q, pUq\}$
  $\{\neg p,\neg q, \neg(pUq)\}\}$

Constructing the automata: transitions

Define the transition relation by setting $s \xrightarrow{a} s'$ iff

1. $X\psi \in s$ implies $\psi \in s'$
2. $\neg X\psi \in s$ implies $\neg\psi \in s'$
3. $\psi_1 U \psi_2 \in s$ and $\psi_2 \notin s$ implies $\psi_1 U \psi_2 \in s'$
4. $\neg(\psi_1 U \psi_2) \in s$ and $\psi_1 \in s$ implies $\neg(\psi_1 U \psi_2) \in s'$
5. a = set of all atomic propositions that hold in s

N.B.: Conditions 3. and 4. are there because

$$\psi_1 U \psi_2 \equiv \psi_2 \vee (\psi_1 \wedge X(\psi_1 U \psi_2))$$

$$\psi_1 R \psi_2 \equiv \psi_2 \wedge (\psi_1 \vee X(\psi_1 R \psi_2))$$

Constructing the automata: acceptance

- For each $\chi_i U \psi_i \in Cl(\phi)$ define the set of accepting states F_i by
  - ☐ $s \in F_i$ iff $\neg(\chi_i U \psi_i) \in s$ or $\psi_i \in s$
  - ☐ The above means that we only accept executions for which infinitely many time $\neg(\chi_i U \psi_i) \vee \psi_i$ holds

- Intuition:

For each $\chi_i U \psi_i \in Cl(\phi)$ we have to guarantee that eventually ψ_i holds.

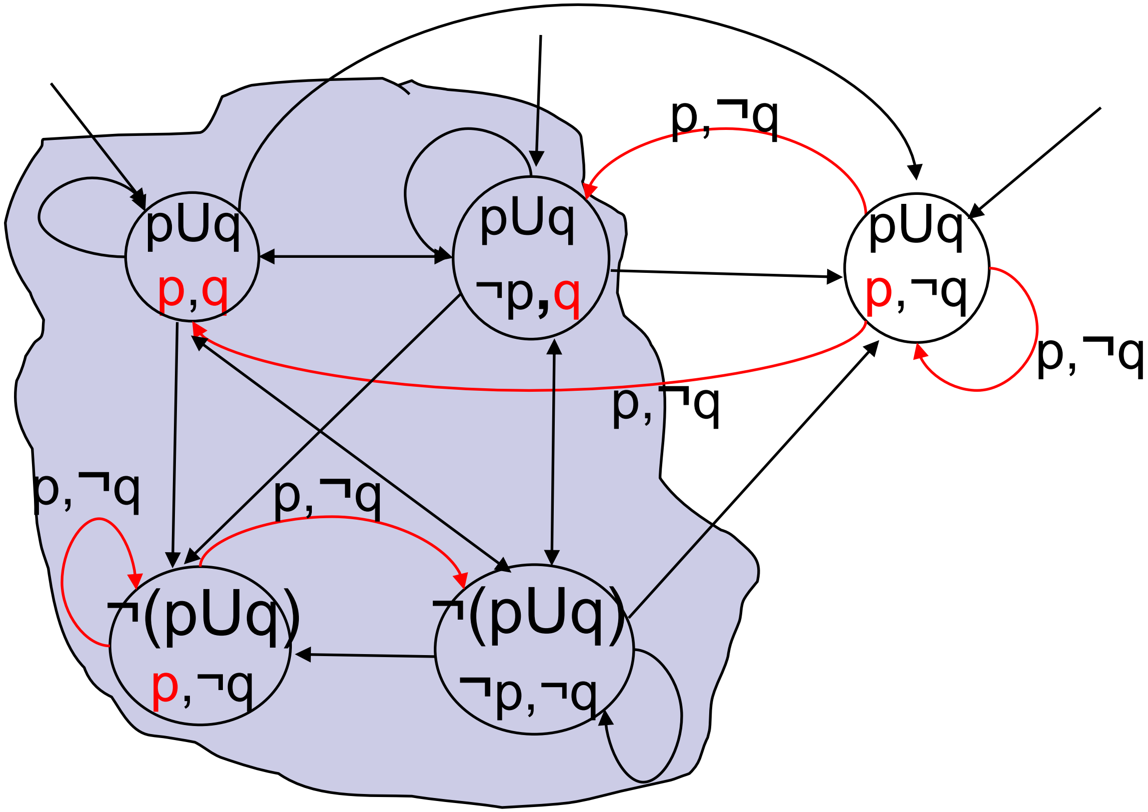
1. Suppose we accept an execution for which only finitely many time $\neg(\chi_i U \psi_i) \vee \psi_i$ holds.
2. Then we can find a suffix such that $\neg(\chi_i U \psi_i) \vee \psi_i$ will never hold, that is $(\chi_i U \psi_i) \wedge \neg\psi_i$ will always hold.
3. Thus we have an execution for which our goal is not guaranteed

Complexity

- $A_{\neg\phi}$ has size $O(2^{|\phi|})$ in the worst case
- The product $A \otimes B$ has size $O(|A| x |B|)$
- We can determine if there no acceptable path in $A \otimes B$ in $O(|A \otimes B|)$ time
- Thus, model checking $M, s \vDash \phi$ can be done in $O(|M| x\ 2^{|\phi|})$ time

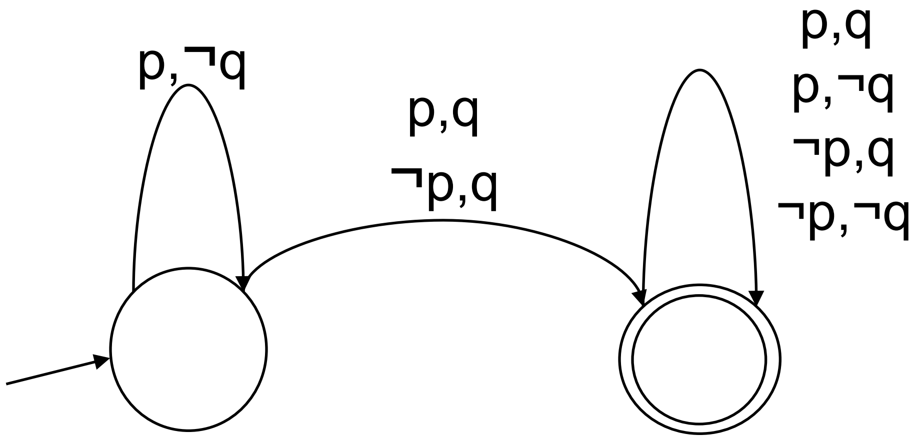
Example: pUq

- Cl(pUq) = { p, ¬p, q, ¬q, pUq, ¬(pUq) }



Example: pUq

- The previous automata is equivalent to

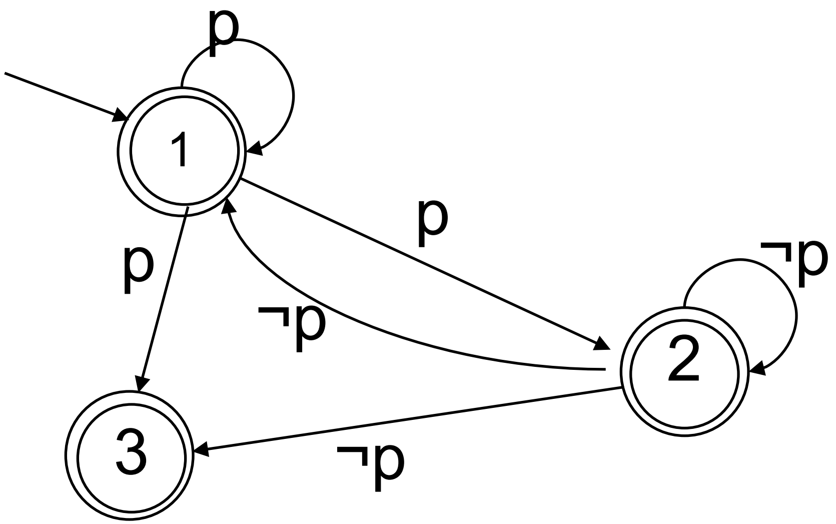


Example II

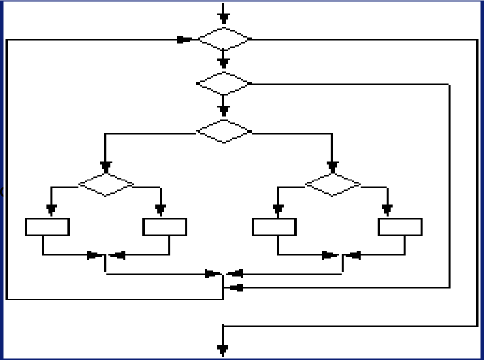
- Buchi automaton for atomic proposition p
  - $p = p \wedge X(T\ U\ T) = \phi$
  - $Cl(\phi) = \{ p, \neg p, T, \neg T, TUT, \neg(T\ U\ T), X(TUT), \neg X(TUT), \phi, \neg\phi\}$
  - $Sub(\phi) = \{1,2,3\}$ with
    - $1 = \{p, T, TUT, X(TUT), \phi \}$,
    - $2 = \{\neg p, T, TUT, X(TUT), \neg\phi\}$
    - $3 = \{p, T, TUT, \neg X(TUT), \neg\phi\}\}$

Example II

- Buchi automaton for atomic proposition p



Spring 2008



Program correctness

Program verification and operational semantics

Marcello Bonsangue

System verification

- Model checking verification is
  - model based $M,s \vDash \phi$
  - fully automatic
  - intended for hardware or software systems with finitely many states
    - control is the main issue
    - no complex data
    - mainly reactive
      - reaction-> computation -> reaction -> …
      - not intended to terminated

System verification

- Program verification:
  - Proof based $\Gamma \vdash \phi$
    - It is impossible to check infinite states !
  - Semi-automatic
  - intended for software systems with possibly infinite states
    - mainly sequential
    - transformational
      - input -> computation -> output
      - like methods of an object

Program verification

The verification framework:

1. Convert an informal specification S in an ‘equivalent’ formula ϕ of some logic
2. Write a program P realizing ϕ (or S)
3. Prove that P satisfies the formula ϕ

A simple language

- Syntactic sets associated to the language:

N	positive and negative integers n,...	
B	truth values	true,false
Var	program variables	x,…
Aexp	arithmetic expressions	a,...
Bexp	boolean expressions	b,...
Com	commands	c, …

Arithmetic expressions

- A ::= n | x | (A+A) | (A-A) | (A*A)

  where $n \in N$ and $x \in Var$
- Here * binds more tightly than - and +
- Examples:

  2 + 3 * 4 - 5 is (2 + 3) * (4 - 5)

  \- 3 is (0 - 3)

  \- -5 is (0 - -5)

  2 + x + 5 is (2 + x) + 5

Boolean expressions

- B ::= true | false | $\neg$B | B$\wedge$B | B$\vee$B | A < A

- Examples:

$A_1 = A_2$	is	$\neg(A_1 < A_2) \wedge \neg(A_2 < A_1)$
$A_1 \neq A_2$	is	$\neg(A_1 = A_2)$

- Boolean expression are built on top of arithmetic expressions
  - 3+5 < 9
  - 4 = 5 is a correct boolean expression !!!
  - true < 10 is not a boolean expression

Commands

- C ::=
```
skip |
x := A |
C;C |
if B then C else C fi |
while B do C od
```

- Example (Fact1)
```
y := 1;
z := 0;
while z ≠ 0 do
          z := z + 1;
           y := y*z
          od
```

The behaviour

- We need a formal model to understand correctly the behavior of a program
- State $\sigma : \mathrm{Var} \to \mathrm{N}$
- An arithmetic expression a in a state σ evaluates to an integer n

$$\underbrace{\langle a, \sigma \rangle}_{\boxed{\text{configuration}}} \to n$$

Evaluating arithmetic expressions

- $\langle n, \sigma\rangle \rightarrow n$
- $\langle x, \sigma\rangle \rightarrow \sigma(x)$
- If n is the sum of n_1 and n_2

$$\frac{\langle a_1, \sigma\rangle \rightarrow n_1 \quad \langle a_2, \sigma\rangle \rightarrow n_2}{\langle a_1 + a_2, \sigma\rangle \rightarrow n}$$

- If n is the subtraction of n_2 from n_1

$$\frac{\langle a_1, \sigma\rangle \rightarrow n_1 \quad \langle a_2, \sigma\rangle \rightarrow n_2}{\langle a_1 - a_2, \sigma\rangle \rightarrow n}$$

- If n is the product of n_1 and n_2

$$\frac{\langle a_1, \sigma\rangle \rightarrow n_1 \quad \langle a_2, \sigma\rangle \rightarrow n_2}{\langle a_1 * a_2, \sigma\rangle \rightarrow n}$$

An Example Derivation

- What is the n such that

$$<(3+4)-(x*2), \sigma> \rightarrow n\ ?$$

Semantics of arithmetic expressions

- Two arithmetic expressions are equivalent if they evaluate to the same value in all states

$$a_1 \approx a_2$$

$$\text{iff}$$

$$(\forall n \in N. \forall \sigma.\ \langle a_1, \sigma\rangle \to n \iff \langle a_2, \sigma\rangle \to n)$$

- Examples:
  - ☐ $\langle 2+3,\sigma\rangle \to 5$ and $\langle 3+2, \sigma\rangle \to 5$ thus $(2+3) \approx (3+2)$
  - ☐ 2+x is not equivalent to 2+3 because there are states in which x evaluates to an integer different from 3

Evaluating Boolean expressions

- $\langle \text{true}, \sigma \rangle \rightarrow T$
- $\langle \text{false}, \sigma \rangle \rightarrow F$
- $\dfrac{\langle b, \sigma \rangle \rightarrow T}{\langle \neg b, \sigma \rangle \rightarrow F} \qquad\qquad \dfrac{\langle b, \sigma \rangle \rightarrow F}{\langle \neg b, \sigma \rangle \rightarrow T}$
- $\dfrac{\langle b_1, \sigma \rangle \rightarrow t_1 \quad \langle b_2, \sigma \rangle \rightarrow t_2}{\langle b_1 \wedge b_2, \sigma \rangle \rightarrow t}$

where $t = T$ if both $t_1 = T$ and $t_2 = T$, otherwise $t = F$

Evaluating boolean expressions

- $$\frac{\langle b_1, \sigma\rangle \to t_1 \qquad \langle b_2, \sigma\rangle \to t_2}{\langle b_1 \vee b_2, \sigma\rangle \to t}$$

  where $t = T$ if $t_1 = T$ or $t_2 = T$, and $t = F$ otherwise

- If n_1 is less than n_2 then

  $$\frac{\langle a_1, \sigma\rangle \to n_1 \qquad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 < a_2, \sigma\rangle \to T}$$

- If n_1 is greater than or equal to n_2 then

  $$\frac{\langle a_1, \sigma\rangle \to n_1 \qquad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 < a_2, \sigma\rangle \to F}$$

Semantics of Boolean expressions

- Two Boolean expressions are equivalent if they evaluate to the same truth value in all states

$$b_1 \approx b_2$$

iff

$$(\forall\sigma.\ \langle b_1, \sigma\rangle \to T \ \Leftrightarrow\ \langle b_2, \sigma\rangle \to T)$$

- We could improve the evaluation of Boolean expressions using
  - ☐ a left-first sequential strategy
  - ☐ a parallel strategy

The command behaviour

- A program may
  - ☐ terminate in a final state or
  - ☐ diverge and never yield a final state

- We denote by

  $$<c, \sigma> \rightarrow \sigma'$$

  the execution of a command c in an initial state σ and terminating in a final state σ'

- Recall: $\sigma[n/x](y) = \begin{cases} n & \text{if } x = y \\ \sigma(y) & \text{if } x \neq y \end{cases}$

Executing commands I

- $<\mathrm{skip}, \sigma> \to \sigma$
- $$\frac{<a, \sigma> \to n}{<x := a, \sigma> \to \sigma[n/x]}$$
- $$\frac{<c_1, \sigma> \to \sigma'' \qquad <c_2, \sigma''> \to \sigma'}{<c_1; c_2, \sigma> \to \sigma'}$$
- $$\frac{<b, \sigma> \to T \quad <c_1, \sigma> \to \sigma'}{<\underline{\mathrm{if}}\ b\ \underline{\mathrm{then}}\ c_1\ \underline{\mathrm{else}}\ c_2\ \underline{\mathrm{fi}}, \sigma> \to \sigma'}$$
- $$\frac{<b, \sigma> \to F \quad <c_2, \sigma> \to \sigma'}{<\underline{\mathrm{if}}\ b\ \underline{\mathrm{then}}\ c_1\ \underline{\mathrm{else}}\ c_2\ \underline{\mathrm{fi}}, \sigma> \to \sigma'}$$

Example: MAX

- What is the final state σ’ of

  <if x < y then z:=y else z:= x fi, σ> → σ’

  for σ(x) = 2, σ(y) = 1 and σ(z) = 0 ?

Executing commands II

- $$\frac{\langle b, \sigma\rangle \to F}{\langle \underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma\rangle \to \sigma}$$

- $$\frac{\langle b, \sigma\rangle \to T \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle \underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma''\rangle \to \sigma'}{\langle \underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma\rangle \to \sigma'}$$

Semantics of commands

- Two commands are equivalent if when executed from the same initial state they terminate in the same final state

$$c_1 \approx c_2$$

iff

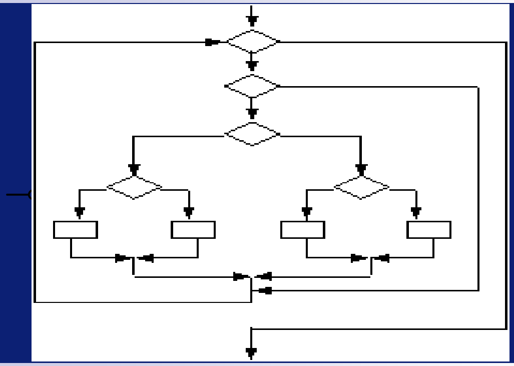
$$(\forall \sigma, \sigma'.\ <c_1, \sigma> \to \sigma' \ \Leftrightarrow\ <c_2, \sigma> \to \sigma')$$

- Examples
  - x := x ≈ skip
  - while b do c of ≈ if b then c; while b do c od
    else skip
    fi

Execution of Commands

- The order of evaluation is important and explicit.
  - ☐ c_1 is evaluated before c_2 in c_1; c_2
  - ☐ c_2 is not evaluated in <u>if</u> true <u>then</u> c_1 <u>else</u> c_2 <u>fi</u>
  - ☐ b is evaluated first in <u>if</u> b <u>then</u> c_1 <u>else</u> c_2 <u>fi</u>
  - ☐ c is not evaluated in "<u>while</u> false <u>do</u> c <u>od</u>
- The execution rules suggest an interpreter but abstract from a concrete one
- Execution is deterministic: only one rule can be applied at time.

Spring 2008



Program correctness

Axiomatic semantics

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

Axiomatic Semantics

- We have introduced
    - a syntax for sequential programs
    - An operational semantics (transition system) for “running” those programs from a starting state. A computation may terminate in a state or run forever.

- We would also like to have a semantics for reasoning about program correctness

Axiomatic semantics

- We need
  - A logical language for making assertions about programs
    - The program terminates
    - If x = 0 then y = z+1 throughout the rest of the execution of the program
    - If the program terminates, then x = y + z
  - A proof system for establishing those assertions

Why axiomatic semantics

- Documentation of programs and interfaces (Meyer's Design by Contract)
- Guidance in language design and coding
- Proving the correctness of algorithms
- Extended static checking
  - checking array bounds
- Proof-carrying code

- Why not testing?
  - Dijkstra: *Program testing can be used to show the presence of bugs, but never to show their absence!*

The idea

“*Compute a number y whose square is less than the input x*”

We have to write a program P such that

y*y < x

But what if x = -4?

There is no program computing y!!

The idea (continued)

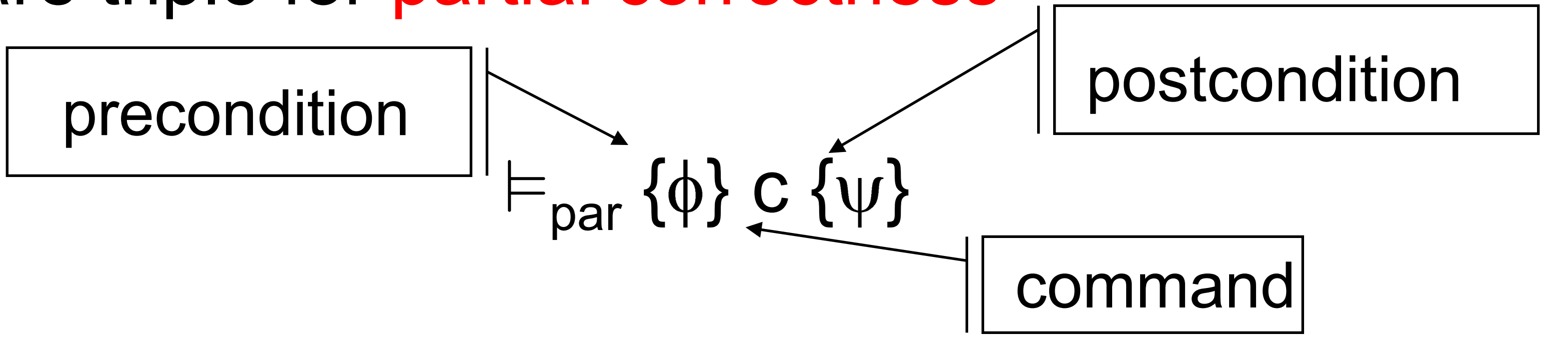
“If the input x is a positive number then compute a number y whose square is less than the input x”

We need to talk about the states before and after the execution of the program P

$$\{ x>0 \}\ P\ \{ y*y < x \}$$

The idea (continued)

- Hoare triple for partial correctness



$$\models_{par} \{\phi\}\ c\ \{\psi\}$$

If the command c terminates when it is executed in a state that satisfies ϕ, then the resulting state will satisfy ψ

program termination is not required

Examples

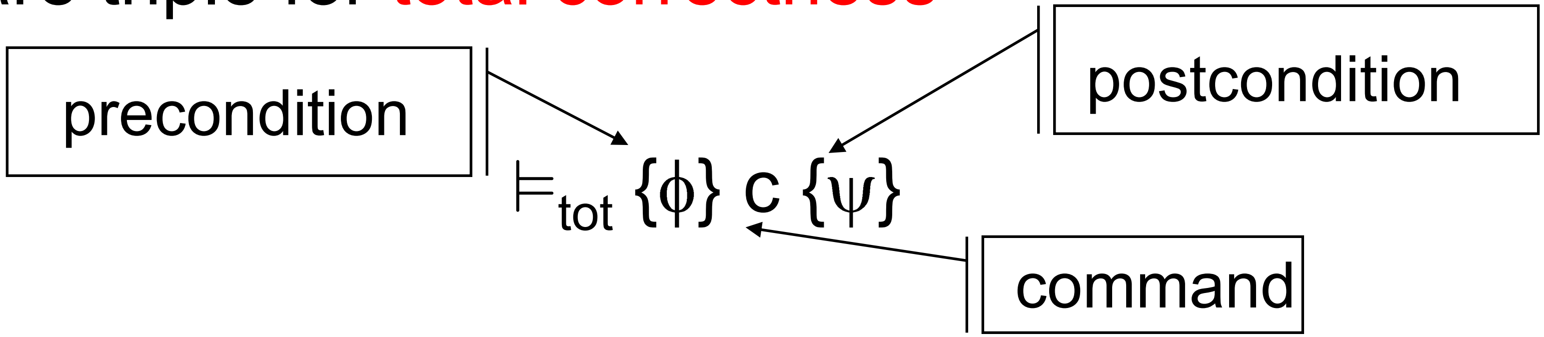
- $\models_{par}\{ y \leq x \}$ z := x; z := z +1 $\{ y < z \}$ is valid
- $\models_{par}\{$ true $\}$ while true do skip od $\{$ false $\}$ is valid
- Let Fact = y := 1; z := 0;

```
        while z ≠ x do
              z := z + 1;
              y := y*z
        od
```

Is $\models_{par} \{ x \geq 0 \}$ Fact $\{ y = x! \}$ valid?

Total correctness

- Hoare triple for total correctness



precondition

postcondition

$\models_{tot} \{\phi\}\ c\ \{\psi\}$

command

If the command c is executed in a state that satisfies ϕ then c is guaranteed to terminate and the resulting state will satisfy ψ

program termination is required

Example

- $\models_{tot} \{ y \leq x \}\ z := x;\ z := z + 1\ \{ y < z \}$ is valid
- $\models_{tot} \{ \text{true} \}$ <u>while</u> true <u>do</u> skip <u>od</u> $\{ \text{false} \}$ is not valid
- $\models_{tot} \{ \text{false} \}$ <u>while</u> true <u>do</u> skip <u>od</u> $\{ \text{true} \}$ is valid
- Let Fact = y := 1; z := 0;

```
while $z \neq x$ do
        z := z + 1;
        y := y*z
od
```

Is $\models_{tot} \{ x \geq 0 \}$ Fact $\{ y = x! \}$ valid?

Partial and total correctness: meaning

- Hoare triple for partial correctness $\vDash_{par} \{\phi\}\ c\ \{\psi\}$

  If ϕ holds in a state σ and $<c,\sigma> \rightarrow \sigma'$ then
  ψ holds in σ'

- Hoare triple for total correctness $\vDash_{tot} \{\phi\}\ c\ \{\psi\}$

  If ϕ holds in a state σ then
  there exists a σ' such that $<c,\sigma> \rightarrow \sigma'$ and ψ holds in σ'

- To be more precise, we need to:
  - ☐ Formalize the language of assertions for ϕ and ψ
  - ☐ Say when an assertion holds in a state.
  - ☐ Give rules for deriving Hoare triples

The assertion language

- Extended arithmetic expressions

$$a ::= n \mid x \mid i \mid (a+a) \mid (a-a) \mid (a*a)$$

$n \in N$, $x \in Var$, $i \in LVar$

- Assertions (or extended Boolean expressions)

$$\phi ::= true \mid \neg\phi \mid \phi\wedge\phi \mid a < a \mid \forall i.\phi$$

$i \in LVar$

Program variables

- We need program variables Var in our assertion language
  - ☐ To express properties of a state of a program as basic assertion such as

$x = n$ i.e. “*The value of x is* n”

that can be used in more complex formulas such as

$x = n \Rightarrow y+1 = x*(y-x)$ i.e. “*If the value of x is* n *then that of y* + 1 *is x times y* - *x*”

Logical variables

- We need a set of logical variables LVar
  - ☐ To express mathematical properties such as

    $\exists i.\ n = i * m$ i.e. "*an integer* n *is multiple of another* m"

  - ☐ To remember the value of a program variable destroyed by a computation

```
Fact2   ≡   y := 1;
            while x ≠ 0 do
                y := y*x;
                x := x – 1
            od
```

$\models_{par} \{\ x \geq 0\ \}$ Fact2 $\{\ y = x!\ \}$ is not valid but

$\models_{par} \{\ x = x_0 \wedge x \geq 0\ \}$ Fact2 $\{\ y = x_0!\ \}$ is.

Meaning of assertions

- Next we assign meaning to assertions
  - Problem: "ϕ holds in a state σ" may depends on the value of the logical variables in ϕ
  - Solution: use interpretations of logical variables
  - Examples
    - $z < x$ holds in a state $\sigma : \mathrm{Var} \to \mathrm{N}$ with $\sigma(x) = 3$ for all interpretations $I : \mathrm{LVar} \to \mathrm{N}$ of the logical variables such that $I(i) < 3$
    - $i < i+1$ holds in a state for all interpretations

Meaning of expressions

- Given a state $\sigma : Var \rightarrow N$ and an interpretation $I : LVar \rightarrow N$ we define the meaning of an expression e as $[[e]]I\sigma$, inductively given by
  - $[[n]]I\sigma = n$
  - $[[x]]I\sigma = \sigma(x)$
  - $[[i]]I\sigma = I(i)$
  - $[[a_1+a_2]]I\sigma = [[a_1]]I\sigma + [[a_2]]I\sigma$
  - $[[a_1-a_2]]I\sigma = [[a_1]]I\sigma - [[a_2]]I\sigma$
  - $[[a_1*a_2]]I\sigma = [[a_1]]I\sigma * [[a_2]]I\sigma$

Meaning of assertions

- Given a state $\sigma : Var \rightarrow N$ and an interpretation $I : LVar \rightarrow N$ we define

  $$\sigma, I \vDash \phi$$

  inductively by

  - $\sigma, I \vDash true$
  - $\sigma, I \vDash \neg\phi$ iff not $\sigma, I \vDash \phi$
  - $\sigma, I \vDash \phi \wedge \psi$ iff $\sigma, I \vDash \phi$ and $\sigma, I \vDash \psi$
  - $\sigma, I \vDash a_1 < a_2$ iff $[[a_1]]I\sigma < [[a_2]]I\sigma$
  - $\sigma, I \vDash \forall i.\phi$ iff $\sigma, I[n/i] \vDash \phi$ for all $n \in N$

Partial and total correctness

- Partial correctness: $I \vDash_{par} \{\phi\}\ c\ \{\psi\}$

$$\forall\sigma\ (\sigma, I \vDash \phi \text{ and } \langle c, \sigma\rangle \to \sigma'\ ) \Rightarrow \sigma', I \vDash \psi$$

- Total correctness: $I \vDash_{tot} \{\phi\}\ c\ \{\psi\}$

$$\forall\sigma.\ \sigma, I \vDash \phi \Rightarrow \exists\ \sigma'.(\langle c, \sigma\rangle \to \sigma' \text{ and } \sigma', I \vDash \psi)$$

where ϕ and ψ are assertions and c is a command

Validity

- To give an absolute meaning to

  $$\{i < x\}\ x := x+3\ \{i < x\}$$

  we have to quantify over all interpretations I

- Partial correctness:

$$\vDash_{par} \{\phi\}\ c\ \{\psi\} \quad \equiv \quad \forall I.\ I \vDash_{par} \{\phi\}\ c\ \{\psi\}$$

- Total correctness:

$$\vDash_{tot} \{\phi\}\ c\ \{\psi\} \quad \equiv \quad \forall I.\ I \vDash_{tot} \{\phi\}\ c\ \{\psi\}$$

Deriving assertions

- We have the meaning of both

  $$\vDash_{par}\{\phi\}\ c\ \{\psi\} \quad \text{and} \quad \vDash_{tot}\{\phi\}\ c\ \{\psi\}$$

  but it depends on the operational semantics and it cannot be effectively used
- Thus we want to define a proof system to derive symbolically valid assertions from valid assertions.
  - ☐ $\vdash_{par}\{\phi\}c\{\psi\}$ means that the Hoare triple $\{\phi\}c\{\psi\}$ can be derived by some axioms and rules
  - ☐ Similarly for $\vdash_{tot}\{\phi\}c\{\psi\}$

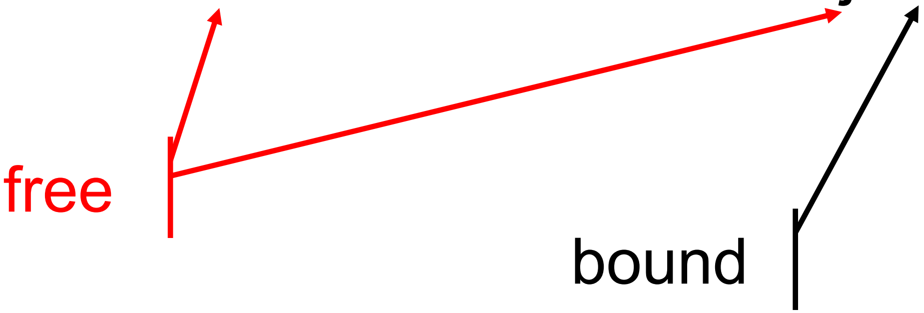
Free and bound variables

- A logical variable is bound in an assertion if it occurs in the scope of a quantifier

$$\exists i.\ n = i * m$$

- A logical variable is free if it is not bound

$$i + 100 < 77 \wedge \forall\, i.\ j+i = 3$$



free

bound

Substitution (I)

- For an assertion ϕ, logical variable i and arithmetic expression e we define

  $$\phi[e/i]$$

  as the assertion resulting by substituting in ϕ the free occurrence of i by e.

- Definition for extended arithmetic expressions

$n[e/i] = n$

$x[e/i] = x$

$i[e/i] = e$

$j[e/i] = j$

$(a_1+a_2)[e/i]=(a_1[e/i]+a_2[e/i])$

$(a_1-a_2)[e/i]=(a_1[e/i]-a_2[e/i])$

$(a_1*a_2)[e/i]=(a_1[e/i]*a_2[e/i])$

Substitution (II)

- Definition for assertions

$$\begin{aligned}
&\text{true}[e/i] && = \text{true} \\
&(\neg\phi)[e/i] && = \neg(\phi[e/i]) \\
&(\phi_1 \wedge \phi_2)[e/i] && = (\phi_1[e/i] \wedge \phi_2[e/i]) \\
&(a_1 < a_2)[e/i] && = (a_1[e/i] < a_2[e/i]) \\
&(\forall i.\phi)[e/i] && = \forall i.\phi \\
&(\forall j.\phi)[e/i] && = \forall j.\phi[e/i] \qquad j \neq i
\end{aligned}$$

- Pictorially, if ϕ = ---i--i--i- with i free, then

  $\phi[e/i]$ = ---e--e--e-

Proof rules partial correctness (I)

- There is one derivation rule for for each command in the language.
  - $\{\phi\}$ skip $\{\phi\}$ skip
  - $\{\phi[a/x]\}$ x := a $\{\phi\}$ ass
  - $$\frac{\{\phi\}\ c_1\ \{\psi\} \qquad \{\psi\}\ c_2\ \{\varphi\}}{\{\phi\}\ c_1;\ c_2\ \{\varphi\}}$$ seq

Proof rules partial correctness (II)

- $$\frac{\{\phi \wedge b\}\ c_1\ \{\psi\} \qquad \{\phi \wedge \neg b\}\ c_2\ \{\psi\}}{\{\phi\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2\ \underline{\text{fi}}\ \{\psi\}} \quad \text{if}$$

- $$\frac{\{\phi \wedge b\}\ c\ \{\phi\}}{\{\phi\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}\ \{\phi \wedge \neg b\}} \quad \text{while}$$

- $$\frac{\vdash \phi \Rightarrow \phi' \qquad \{\phi'\}\ c\ \{\psi'\} \qquad \vdash \psi' \Rightarrow \psi}{\{\phi\}\ c\ \{\psi\}} \quad \text{cons}$$

A first example: assignment

- Let’s prove that

$$\vdash_{par} \{\text{true}\}\ x:=1\ \{x=1\}$$

$$\dfrac{\vdash \text{true} \Rightarrow 1=1 \qquad \dfrac{}{\{1=1\}\ x:=1\ \{x=1\}}\ \text{ass}}{\{\text{true}\}\ x:=1\ \{x=1\}}\ \text{cons}$$

Another example: assignment

- Prove that {true} x:= e {x=e} when x does not appear in e

1. Because x does ot appear in e we have

$$(x=e)[e/x] \;\equiv\; (x[e/x]=e[e/x]) \;\equiv\; (e=e)$$

2. Use assignment + consequence to obtain the proof

$$\dfrac{\vdash \text{true} \Rightarrow e=e \qquad \dfrac{}{\{e=e\}\ x:=e\ \{x=e\}}\,\text{ass}}{\{\text{true}\}\ x:=e\ \{x=e\}}\,\text{cons}$$

Another example: conditional

- Prove $\vdash_{par} \{true\}\ \underline{if}\ y \leq 1\ \underline{then}\ x:=1\ \underline{else}\ x:=y\ \underline{fi}\ \{x>0\}$

---------------------- ass   ass ----------------------

$\vdash true \wedge y \leq 1 \Rightarrow 1>0$ $\{1>0\}\ x:=1\ \{x>0\}$ $\vdash true \wedge y>1 \Rightarrow y>0$ $\{y>0\}\ x:=y$
$\{x>0\}$

-- cons --

$\{true \wedge y \leq 1\}\ x:=1\ \{x>0\}$ $\{true \wedge y >\}\ x:=y\ \{x>0\}$

-- if

$\{true\}\ \underline{if}\ y \leq 1\ \underline{then}\ x:=1\ \underline{else}\ x:=y\ \underline{fi}\ \{x>0\}$

An example: while

- Prove $\vdash_{par} \{0 \leq x\}\ \underline{\text{while}}\ x>0\ \underline{\text{do}}\ x:=x-1\ \underline{\text{od}}\ \{x=0\}$

  We take as invariant $0 \leq x$ in the while-rule

------------------------------ ass

$\vdash 0 \leq x \wedge x>0 \Rightarrow 0 \leq x-1 \qquad \{0 \leq x-1\}\ x:=x-1\ \{0 \leq x\}$

-- cons

$\{0 \leq x \wedge x>0\}\ x:=x-1\ \{0 \leq x\}$

--while

$\{0 \leq x\}\ \underline{\text{while}}\ x>0\ \underline{\text{do}}\ x:=x-1\ \underline{\text{od}}\ \{0 \leq x \wedge x \leq 0\} \qquad \vdash 0 \leq x \wedge x \leq 0 \Rightarrow x=0$

--- cons

$\{x \leq 0\}\ x>0\ \underline{\text{do}}\ x:=x-1\ \underline{\text{od}}\ \{x=0\}$

An example: while, again

Prove that $\{x \le 0\}\ \underline{\text{while}}\ x \le 5\ \underline{\text{do}}\ x:=x+1\ \underline{\text{od}}\ \{x=6\}$

1. We start with the invariant $x \le 6$ in the while-rule

$$\cfrac{\cfrac{\vdash x \le 6 \wedge x \le 5 \Rightarrow x+1 \le 6 \qquad \cfrac{}{\{x+1 \le 6\}x:=x+1\ \{x \le 6\}}\ \text{ass}}{\{x \le 6 \wedge x \le 5\}\ x:=x+1\ \{x \le 6\}}\ \text{cons}}{\{x \le 6\}\ \underline{\text{while}}\ x \le 5\ \underline{\text{do}}\ x:=x+1\ \underline{\text{od}}\ \{x \le 6 \wedge x>5\}}\ \text{while}$$

2. We finish with the consequence rule

$$\cfrac{\vdash x\le0 \Rightarrow x\le6 \quad \{x \le 6\}\ \underline{\text{while}}\ x \le 5\ \underline{\text{do}}\ x:=x+1\ \underline{\text{od}}\ \{x \le 6 \wedge x>5\} \quad \vdash x\le6 \wedge x>5 \Rightarrow x=6}{\{x \le 0\}\ \underline{\text{while}}\ x \le 5\ \underline{\text{do}}\ x:=x+1\ \underline{\text{od}}\ \{x=6\}}$$

Auxiliary rules

- They can be derived from the previous ones
  - $\{\phi\}\ c\ \{\phi\}$ if the program variables in ϕ do not appear in c
  - $\{\phi\}\ x := a\ \{\exists x_0.(\phi[x_0/x] \wedge x = a[x_0/x])\}$
  - $$\frac{\{\phi_1\}\ c_1\ \{\psi\} \qquad \{\phi_2\}\ c_2\ \{\psi\}}{\{(b \Rightarrow \phi_1) \wedge (\neg b \Rightarrow \phi_2)\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2\ \underline{\text{fi}}\ \{\psi\}}$$
  - $$\frac{\{\phi_1\}\ c\ \{\psi\} \qquad \{\phi_2\}\ c\ \{\psi\}}{\{\phi_1 \vee \phi_2\}\ c\ \{\psi\}}$$
  - $$\frac{\{\phi_1\}\ c\ \{\psi_1\} \qquad \{\phi_2\}\ c\ \{\psi_2\}}{\{\phi_1 \wedge \phi_2\}\ c\ \{\psi_1 \wedge \psi_2\}}$$

Comments on Hoare logic

- The rules are syntax directed
  - Three problems:
    - When to apply the consequence rule
    - How to prove the implication in the consequence rule
    - What invariant to use in the while rule
- The last is the real hard one
  - Should it be given by the programmer?

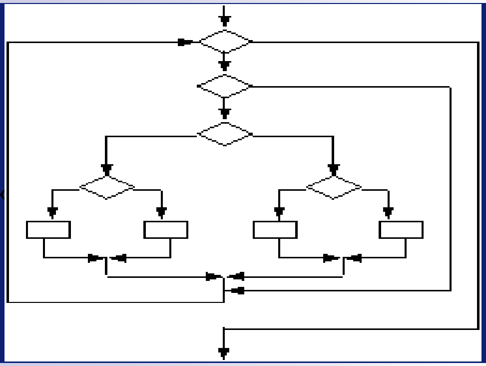
An extensive example: a program

```
DIV ≙
    q := 0;
    r := x;
    while r ≥ y do
            r := r-y;
            q := q+1
    od
```

We wish to prove

$\{x \geq 0 \wedge y > 0\}\ \mathrm{DIV}\ \{q*y+r=x \wedge 0 \leq r<y\}$

Spring 2006



Program correctness

Weakest preconditions

Marcello Bonsangue

Axiomatic semantics

- We have a language for asserting properties of programs (syntax).
- We know when an assertion is true (validity).
- We have a symbolic way for deriving assertions (proof system).
- What is the relation between validity and provability?

Hoare Logic
soundness and completeness

- Soundness (what can be proved is valid):

$$\vdash_{par} \{\phi\}\ c\ \{\psi\} \quad \text{implies} \quad \models_{par} \{\phi\}\ c\ \{\psi\}$$

- Completeness (what is valid can be proved):

$$\models_{par} \{\phi\}\ c\ \{\psi\} \quad \text{implies} \quad \vdash_{par} \{\phi\}\ c\ \{\psi\}$$

Soundness

- Theorem: The proof system for partial correctness is sound

  equivalently, if $\vdash_{par} \{\phi\}\ c\ \{\psi\}$ then

  $\forall \sigma, I \ \ (\sigma, I \ \vDash_{par} \phi \text{ and } \langle c, \sigma\rangle \rightarrow \sigma' \,) \Rightarrow \sigma', I \vDash_{par} \psi$

  Proof by induction on the length of the derivation of the Hoare triples, reasoning about each axiom and rule separately. (why?)

Soundness of skip

Case: last rule used in the derivation is

$$\{\phi\}\ \underline{\text{skip}}\ \{\phi\}.$$

We have to prove

$$\forall \sigma, l\ (\sigma, l \vDash_{par} \phi \text{ and } \langle \underline{\text{skip}}, \sigma\rangle \to \sigma') \Rightarrow \sigma', l \vDash_{par} \phi$$

Which follows because $\sigma' = \sigma$.

Soundness of assignment

Case last rule in the derivation is $\{\phi[a/x]\}\ x := a\ \{\phi\}$

Take σ and I such that $\sigma, I \vDash \phi[a/x]$. Then

$$< x := a, \sigma > \to \sigma[a/x]$$

We need to prove $\sigma[a/x], I \vDash \phi$, which follows from the substitution lemma

LEMMA: $\sigma, I \vDash \phi[a/x]$ implies $\sigma[a/x], I \vDash \phi$

Proof: by induction on the structure of ϕ

Soundness of consequence rule

- Case last rule in the derivation is

$$\frac{\vdash \phi \Rightarrow \phi' \qquad \{\phi'\}\ c\ \{\psi'\} \qquad \vdash \psi' \Rightarrow \psi}{\{\phi\}\ c\ \{\psi\}}$$

- From soundness of first order logic we have

$$\sigma, I \models \phi \Rightarrow \phi'.$$

  Hence $\sigma, I \models \phi'$.

- From induction hypothesis we get $\sigma', I \models \psi'$.

- From soundness of first order logic we finally obtain

$$\sigma', I \models \psi' \Rightarrow \psi\ .$$

  Therefore $\sigma', I \models \psi$

Soundness of while

- Case last rule in the derivation is

$$\frac{\{\phi \wedge b\}\ c\ \{\phi\}}{\{\phi\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}\ \{\phi \wedge \neg b\}}$$

- Assume $\sigma, I \vDash \phi$. We proceed by induction on the derivation of $<\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma> \rightarrow \sigma'$
  - There are two cases (we treat only one):

$$\frac{<b, \sigma> \rightarrow T \qquad <c, \sigma> \rightarrow \sigma' \quad <\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma'> \rightarrow \sigma''}{<\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma> \rightarrow \sigma''}$$

  - We need to prove $\sigma'', I \vDash \phi \wedge \neg b$

Soundness of while (II)

- By definition of derivation of $<b, \sigma> \to T$ we obtain

  $$\sigma, I \vDash b$$

  Hence $\sigma, I \vDash \phi \wedge b$

- By induction hypothesis on derivation of $\{\phi \wedge b\}$ c $\{\phi\}$ we have

  $$\sigma', I \vDash \phi$$

- By induction hyp. on derivation of $<\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma'> \to \sigma''$ we finally obtain

  $$\sigma'', I \vDash \phi \wedge \neg b$$

Hoare Logic

- We have seen that if we can derive an assertion in the Hoare logic then this assertion is true (soundness).
- Next we concentrate on the opposite direction (completeness).

Completeness of Hoare Logic

- Can we prove that if an assertion is true then it is derivable?
- More formally, can we prove

$$\vDash_{par}\{\phi\}\ c\ \{\psi\} \quad \text{implies} \ \vdash_{par}\{\phi\}\ c\ \{\psi\}?$$

- The answer is yes, but only if the underlying logic is complete ($\vDash \phi$ implies $\vdash \phi$) and expressive enough
  - ☐ This is called relative completeness.

Idea for proving completeness

- To prove $\models_{tot}\{\phi\}\ c\ \{\psi\}$ implies $\vdash_{tot}\{\phi\}\ c\ \{\psi\}$

1. Assume we can compute $wp(c,\psi)$ such that
   - $wp(c,\psi)$ is a precondition of ψ, i.e.

     $$\vdash_{tot} \{wp(c,\psi)\}\ c\ \{\psi\}$$

   - $wp(c,\psi)$ is the weakest precondition of ψ, i.e.

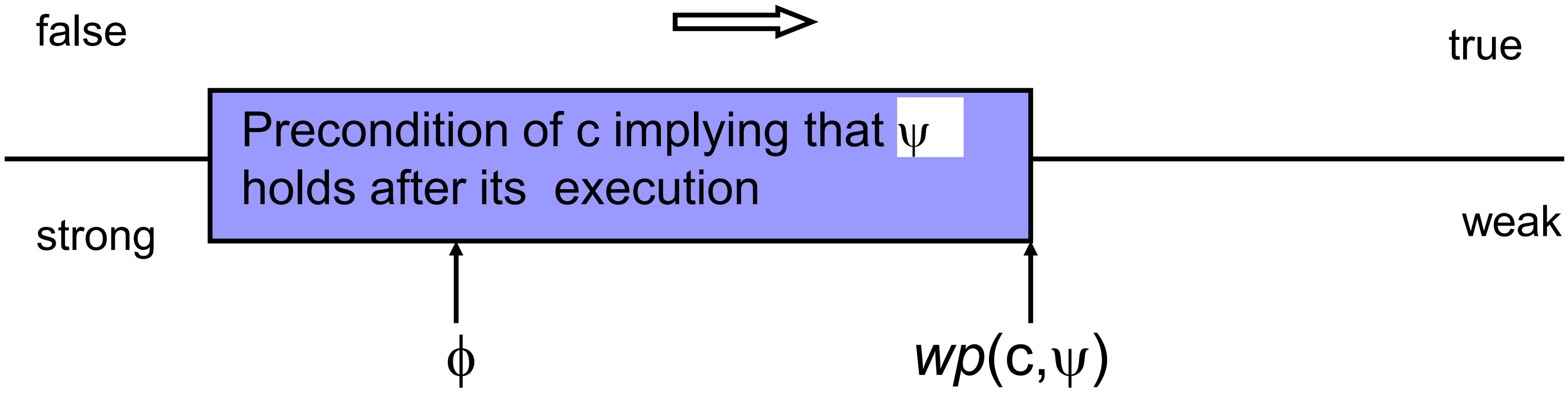
     $$\models_{tot}\{\varphi\}\ c\ \{\psi\} \text{ implies } \models \varphi \Rightarrow wp(c,\psi)$$

2. By completeness of the underlying logic and the consequence rule we obtain

$$\frac{\vdash \phi \Rightarrow wp(c,\psi) \qquad\qquad \vdash_{tot} \{wp(c,\psi)\}\ c\ \{\psi\}}{\vdash_{tot} \{\phi\}\ c\ \{\psi\}}$$

Weakest precondition (Dijkstra)

- Assertions can be ordered



false

$\Rightarrow$

true

Precondition of c implying that ψ holds after its execution

strong

weak

ϕ

$wp(c,\psi)$

- Thus to verify $\{\phi\}$ c $\{\psi\}$ we compute $wp(c,\psi)$ and prove $\phi \Rightarrow wp(c,\psi)$

Weakest precondition

- The definition of the weakest precondition follows the rules of the Hoare logic

- SKIP

$$\frac{}{\{\phi\}\ \text{skip}\ \{\phi\}}$$

$$wp(\text{skip},\phi) = \phi$$

Weakest precondition

- ASSIGNMENT

$$\frac{}{\{\phi[a/x]\}\ x := a\ \{\phi\}}$$

$$wp(x{:=}a,\phi) = \phi[a/x]$$

- SEQUENTIAL COMPOSITION

$$\frac{\{\phi\}\ c_1\ \{\psi\} \qquad \{\psi\}\ c_2\ \{\varphi\}}{\{\phi\}\ c_1;\ c_2\ \{\varphi\}}$$

$$wp(c_1;\ c_2,\ \varphi) = wp(c_1, wp(c_2,\ \varphi))$$

Weakest precondition

- CONDITIONAL

$$\frac{\{\phi_1\}\ c_1\ \{\psi\} \qquad \{\phi_2\}\ c_2\ \{\psi\}}{\{b \Rightarrow \phi_1 \wedge \neg b \Rightarrow \phi_2\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2\ \underline{\text{fi}}\ \{\psi\}}$$

$$wp(\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2\ \underline{\text{fi}},\ \psi) = b \Rightarrow wp(c_1,\psi) \wedge \neg b \Rightarrow wp(c_2,\psi)$$

Weakest precondition

- LOOP

1. We already know that

   $\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}} \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ (c;\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}})\ \underline{\text{else}}\ \text{skip}\ \underline{\text{fi}}$

2. Let $w = \underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}$ and $W = wp(w, \psi)$. We have

$$W = b \Rightarrow wp(c,W) \wedge \neg b \Rightarrow \psi$$

3. This is a recursive equation
   - We know how to solve it
   - We need a complete partial order (cpo) of assertions

A CPO of assertions

- Refinement order:

  $$\phi \leq \psi \text{ iff } \quad \models \psi \Rightarrow \phi$$

  True is the bottom: it does not says much about a state.

- It forms a complete partial order: the least upper bound of every chain $\phi_1 \leq \phi_2 \leq \ldots \leq \phi_n \leq$ is the infinite conjunction $\bigwedge \phi_i$

  where $\sigma, I \models \bigwedge \phi_i$ iff $\sigma, I \models \phi_i$ for all i

Weakest precondition (LOOP)

- Let $F(X) = b \Rightarrow wp(c, X) \wedge \neg b \Rightarrow \psi$.

- Then F is monotone and continuous. Thus it has a least fixed point (the weakest fixed point) and

$$wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi) = \bigwedge F^i(\text{true})$$

- We need an assertion language expressive enough to be able to write $\bigwedge F^i(\text{true})$.

Weakest precondition (LOOP)

- Define a family of preconditions $wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi)_k$ as follows:

  $$wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi)_0 = \neg b \Rightarrow \psi$$

  $$wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi)_{n+1} = \\ b \Rightarrow wp(c, wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi)_n) \wedge \neg b \Rightarrow \psi$$

  Then $wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi) = \bigwedge wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi)_k$

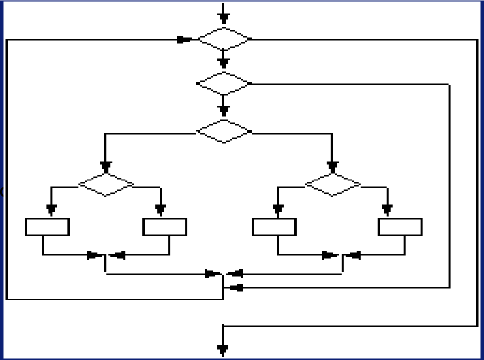
- Here $wp(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \psi)_k$ is the weakest precondition on which the loop - if terminated in k or less iterations - terminates in ψ.

Weakest precondition: properties

- For each command c in our language we have
  - *wp*(c,true) = true
  - if $\psi \Rightarrow \psi'$ then *wp*(c, ψ) $\Rightarrow$ *wp*(c, ψ')
  - *wp*(c, $\psi \wedge \psi'$) = *wp*(c, ψ) $\wedge$ *wp*(c, ψ')
  - *wp*(c, $\psi \vee \psi'$) = *wp*(c, ψ) $\vee$ *wp*(c, ψ')

- *wp*(c,false) characterizes all states in which c does not terminate

Spring 2008



Program correctness

Proof Outlines

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

Proof outlines

- Formal proofs are long and tedious to follow.
- It is better to organize the proof in small local isolated steps
- We can use the structure of the program to structure our proof!

The idea

- For the program $P = c_1; c_2; c_3; \dots c_n$ we want to show

$$\vdash_{par} \{\phi_0\}\ P\ \{\ \phi_n\}$$

- We can split the problem into smaller ones if we find formulas ϕ_i's such that

$$\vdash_{par} \{\phi_i\}\ c_i \{\phi_{i+1}\}$$

The idea (cont.d)

- Thus we have to find a calculus for presenting a proof $\vdash_{par} \{\phi_0\}\ P\ \{\phi_n\}$ by interleaving formulas with code

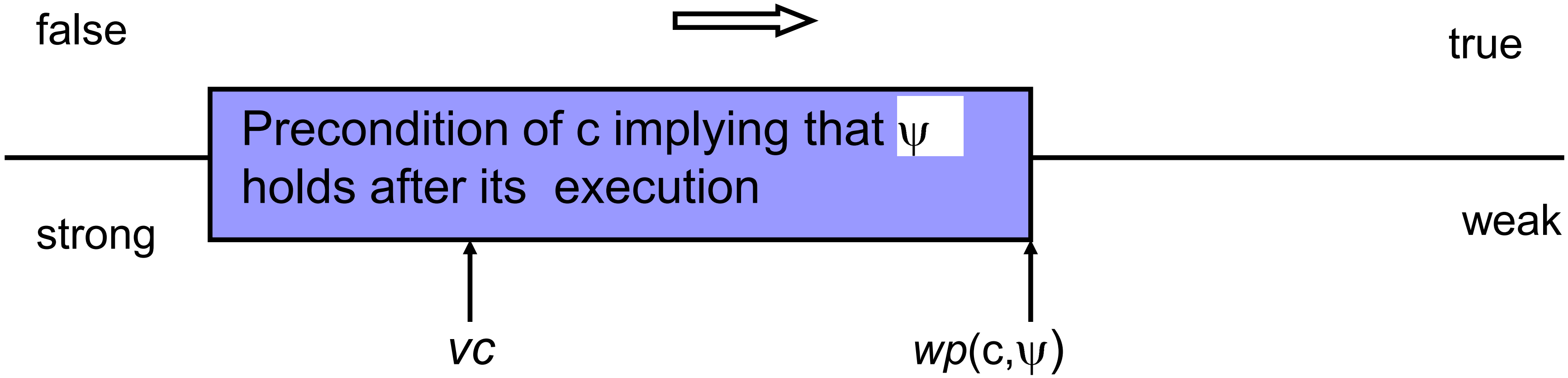
$\{\phi_0\}$
$c_1;$
$\{\phi_1\}$ justification (i.e. skip, ass, if, while, implied)
$c_2;$
$\{\phi_2\}$ justification
$c_3;$
$\vdots$
$\{\phi_{n-1}\}$ justification
c_n
$\{\ \phi_n\}$

Composition is implicit !

Verification condition

Problem: How can we find the ϕ_i's ?

Solution: Use Hoare rules and calculate verification conditions, i.e. conditions needed to establish the validity of certain assertions.



Skip, assignment, implied

- ----------------
  $\{\phi\}$ skip $\{\phi\}$

  skip

- ----------------------
  $\{\phi[a/x]\}$ x := a $\{\phi\}$

  assignment

- $\vdash \phi \Rightarrow \psi$

  $\{\phi\}\ \{\psi\}$

  implied

Example

- To prove $\vdash_{par} \{y = 5\}\ x := y + 1\ \{x = 6\}$

$\{y = 5\}$	
$\{y+1 = 6\}$	implied
$x := y + 1$	
$\{x = 6\}$	assignment

we only need to prove the verification condition $y = 5 \Rightarrow y+1 = 6$

Composition, conditional

- $$\frac{\{\phi\}\ c_1\ \{\psi\} \qquad\qquad \{\psi\}\ c_2\ \{\varphi\}}{\{\phi\}\ c_1;\ \{\ \psi\ \}\ c_2\ \{\varphi\}} \qquad \text{seq}$$

- $$\frac{\{\phi_1\}\ c_1\ \{\psi\} \qquad\qquad \{\phi_2\}\ c_2\ \{\psi\}}{\{b \Rightarrow \phi_1 \wedge \neg b \Rightarrow \phi_2\ \}\underline{\text{if}}\ b\ \underline{\text{then}}\{\phi_1\}c_1\{\psi\}\underline{\text{else}}\{\phi_2\}c_2\{\psi\}\ \text{fi}\{\psi\}} \qquad \text{if}$$

Example

- To prove $\vdash_{par}\{true\}\ z:=x;\ z:=z+y;\ u:=z\ \{u = x+y\}$

$\{true\}$	
$\{ x+y = x+y \}$	implied
$z:=x;$	
$\{ z+y = x+y \}$	assignment
$z:=z+y;$	
$\{ z = x+y \}$	assignment
$u:=z$	
$\{ u = x+y \}$	assignment

we only need to prove the verification condition

$$true \Rightarrow x+y = x+y$$

Example

Suppose we want to prove

{true}
a := x+1;
if a = 1 then y := 1 else y := a fi
{y = x+1}

Example

```
$\{ true \}$
$\{x+1=1 \Rightarrow 1=x+1 \wedge x+1 \neq 1 \Rightarrow x+1=x+1\}$        implied
a := x+1;
$\{a=1 \Rightarrow 1=x+1 \wedge a \neq 1 \Rightarrow a=x+1\}$              assignment
if a = 1
    then   $\{1 = x+1\}$
             y := 1
           $\{ y = x+1\}$                                                  assignment
      else
           $\{a = x+1\}$
             y := a
           $\{ y = x+1 \}$                                                 assignment
fi
$\{ y = x+1 \}$                                                            if-then-else
```

While statement

$$\frac{\{I \wedge b\}\ c\ \{I\}}{\{I\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ \{I \wedge b\}\ c\ \{I\}\ \underline{\text{od}}\ \{I \wedge \neg b\}}\ \text{while}$$

- We must discover an invariant I
  - I need not hold during the execution of c
  - if I holds before c is executed then it holds if and when c terminates.

Invariant

- For any while b do c od these are invariants
  - true
  - false
  - $\neg b$

  because $\{I \wedge b\}\ c\ \{I\}$ is valid. However they are useless to prove

  $$\phi \Rightarrow I \qquad \text{or} \qquad I \wedge \neg b \Rightarrow \psi$$

  when considering the while in a context.
- To find a useful invariant it may help to look at the execution of the while and at the relationships among the variables manipulated by the while-body

Example

- Let W = <u>while</u> $x > 0$ <u>do</u> y := x*y; x := x-1 <u>od</u>
- To prove $\{x = n \wedge n \geq 0 \wedge y=1\}$ W $\{ y = n! \}$

iteration	x	y	$x > 0$?
0	6	1	true
1	5	6	true
2	4	30	true
3	3	120	true
4	2	360	true
5	1	720	true
6	0	720	false

Example I

- Invariant Hypothesis $y*x! = n!$

```
{$y*x! = n!$}
while $x > 0$ do
{ $y*x! = n! \wedge x > 0$ }                invariant and guard
{ $x*y*(x-1)! = n!$ }                       implied
$y := x*y$;
{ $y*(x-1)! = n!$ }                         assignment
$x := x-1$
{ $y*x! = n!$ }                             assignment
od
{ $y*x! = n! \wedge \neg x > 0$ }           while
```

correct !!!

Example II

- Since y*x! = n! is an invariant we have

$\{x = n \wedge n \geq 0 \wedge y=1\}$	
$\{y*x! = n!\}$	implied
W	
$\{y*x! = n! \wedge \neg x > 0\}$	while
$\{y*x! = n! \wedge x \leq 0\}$	implied
$\{y = n!\}$	implied??

The invariant is too weak!

Example III

- Another invariant hypothesis $y*x! = n! \wedge x \geq 0$

```
{$y*x! = n! \wedge x \geq 0$}
while $x > 0$ do
{$y*x! = n! \wedge x \geq 0 \wedge x > 0$}          Inv. Hyp. and guard
{$x*y*(x-1)! = n! \wedge x \geq 1$}                 implied
y := x*y;
{$y*(x-1)! = n! \wedge x-1 \geq 0$}                 assignment
x := x-1
{$y*x! = n! \wedge x \geq 0$}                        assignment
od
{$y*x! = n! \wedge x \geq 0 \wedge \neg x > 0$}     while
```

correct !!!

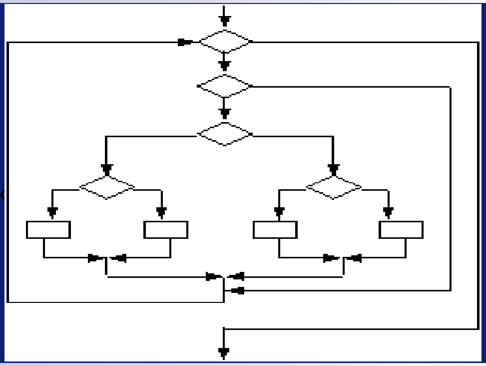
Example IV

- With the new invariant we have

$\{x = n \wedge n \geq 0 \wedge y=1\}$	
$\{y*x! = n! \wedge x \geq 0\}$	implied
W	
$\{y*x! = n! \wedge x \geq 0 \wedge \neg x > 0\}$	while
$\{y*x! = n! \wedge x = 0\}$	implied
$\{y = n!\}$	implied

Yes!

Spring 2008



Program correctness

Arrays

Marcello Bonsangue

Leiden Institute of Advanced Computer Science
Research & Education

Array Types and Array Syntax

- Let a[1 … n] denote an array with as index an integer between 1 and n (included)
- Then a[e] denotes the element at position i in the array a if the evaluation of the expression e is the integer i with $1 \leq i \leq n$
- And |a| denote the length of the array a,
  - i.e. $|a| = n$

Meaning of array assignments

- Let a, b be two array variables. Then:
  - a:=b assigns the value of array a to the array variable b
  - a[e]:=e’ assigns the value of e’ to position e in the array a
  - but a[e]:=e’ fails, or ‘goes wrong’, if $e \leq 0$ or $e < |a|$
- In partial correctness, we do not need to take array boundaries into account
  - For example, {true}a[|a|+1] {true} is valid

Array assignments and aliasing

- Simple assignments remain simple:

  $$\{\psi[b/a]\}\ a:=b\ \{\psi\}$$

  is valid (partial correctness)
- But what about a[e]:=e’ ?
- How can we substitute a[e] by e’ ?
- Moreover, a[e] may have aliases:
  a[3], a[1+2], a[5-2], etc. all denote the same location

Arrays as functions

- An array a[1…|a|] of values can be seen as a function a from the index values to the element values

  update: a[e] := e’ is the same as a :=a[e’/e]

  reading: a[e] is the same as a(e)

- Recall that $a[e'/e](i) = \begin{cases} e' & \text{if } e=i \\ a(i) & \text{otherwise} \end{cases}$

The solution: function substitution

- Since an array is just a variable whose type happens to be “function”, we can simply replace the entire function
- a[i] := e is the same as a := a[e/i] thus along the lines of the ordinary assignment axiom we have

$$\{\psi[a[e'/e]/a]\}\ a[e] := e'\ \{\psi\}$$

Weakest precondition of array updates

- The formula $\psi[a[e'/e]/a]$ is not the weakest precondition of ψ w.r.t. an array update $a[e]:=e'$

  Why?

  Because the value e may fall outside that of the array a, so update may also fail! For total correctness we have to prove that assignment doesn't fail.

- $wp(a[e] := e',\psi)=\psi[a[e'/e]/a] \wedge 0<e\leq|a|$

Example I

- $\{ \text{true} \}\ a[3] := 5\ \{ a[3] = 5 \}$

  We get:

  $(a[3] = 5)[a[5/3]/a] \Leftrightarrow a[5/3][3] = 5$

  Clearly, $\text{true} \Rightarrow a[5/3][3]=5$

Example II

- $\{a[j] = 4\}\ a[i] := a[j]+1\ \{a[i] = 5\}$

$(a[i] = 5)[a[a[j]+1/i]/a]$

$\Leftrightarrow a[a[j]+1/i][i] = 5$

$\Leftrightarrow a[j]+1 = 5$

$\Leftrightarrow a[j] = 4$

Example 3

- $\{|b|>2\}$ a:=b; a[1]:=3; a[1]:= a[1]+1; b:=a $\{b[1]=4\}$

Example 4

$\{ a[i] = i \}\ a[a[i]] := i\ \{ a[i] = i \}$

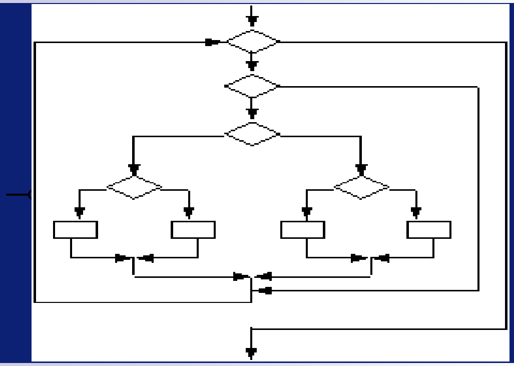
$(a[i] = i)[(a[i/a[i]])/a]$

$\Leftrightarrow a[i/a[i]](i) = i$

$\Leftrightarrow (a[i] = i \wedge i = i) \vee (a[i] \neq i \wedge a[i] = i)$

$\Leftrightarrow a[i] = i$

Spring 2008



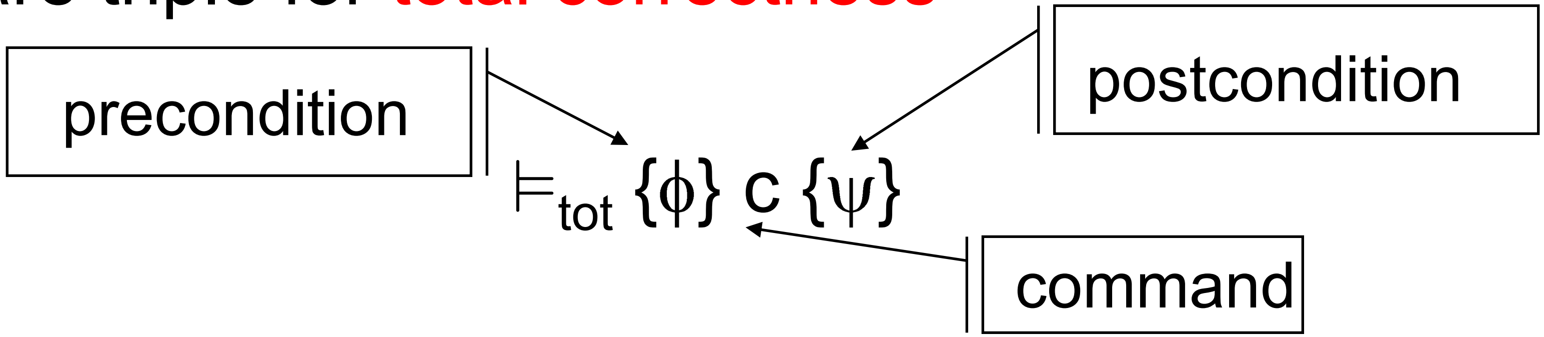
Program correctness

Total correctness

Marcello Bonsangue

Total correctness

- Hoare triple for total correctness



precondition

postcondition

$\models_{tot} \{\phi\}\ c\ \{\psi\}$

command

If the command c is executed in a state that satisfies ϕ then c is guaranteed to terminate and the resulting state will satisfy ψ

program termination is required

Example

- $\models_{tot} \{ y \leq x \}\ z := x;\ z := z + 1\ \{ y < z \}$ is valid
- $\models_{tot} \{ \text{true} \}\ \underline{\text{while}}\ \text{true}\ \underline{\text{do}}\ \text{skip}\ \underline{\text{od}}\ \{ \text{false} \}$ is not valid
- $\models_{tot} \{ \text{false} \}\ \underline{\text{while}}\ \text{true}\ \underline{\text{do}}\ \text{skip}\ \underline{\text{od}}\ \{ \text{true} \}$ is valid
- Let Fact = y := 1; z := 0;

```
while z ≠ x do
        z := z + 1;
        y := y*z
od
```

Is $\models_{tot} \{ x \geq 0 \}\ \text{Fact}\ \{ y = x! \}$ valid?

Total correctness

- Total correctness: $I \vDash_{tot} \{\phi\}\ c\ \{\psi\}$

$\forall\sigma.\ \sigma, I \vDash \phi \Rightarrow \exists\ \sigma'.(<c, \sigma> \rightarrow \sigma'$ and $\sigma', I \vDash \psi)$

where ϕ and ψ are assertions and c is a command

Validity

- To give an absolute meaning to

$$\{i < x\}\ x := x+3\ \{i < x\}$$

  we have to quantify over all interpretations I

- Total correctness:

$$\models_{tot} \{\phi\}\ c\ \{\psi\} \quad \equiv \quad \forall I.\ I \models_{tot} \{\phi\}\ c\ \{\psi\}$$

Towards a calculus

- Partial correctness does not tell anything about termination

- Only while b do c od introduces the possibility of non-termination

⇓

a proof calculus for total correctness is the same as that for partial correctness except for the while-rule

Intuition

- To prove total correctness we need
  - a proof of partial correctness
  - a proof that the while statement terminates

- Termination can be proved by finding an integer expression E (the variant) that
  - is always non-negative
  - decreases every time we execute the body of the while statement

Proof rules
total and partial correctness (I)

- $\{\phi\}$ skip $\{\phi\}$ skip
- $\{\phi[a/x] \wedge \mathrm{def}(a)\}$ x := a $\{\phi\}$ ass
- $$\frac{\{\phi\}\ c_1\ \{\psi\} \qquad \{\psi\}\ c_2\ \{\varphi\}}{\{\phi\}\ c_1;\ c_2\ \{\varphi\}}\quad \text{seq}$$

Proof rules total and partial correctness (II)

- $$\frac{\{\phi \wedge b\}\ c_1\ \{\psi\} \qquad \{\phi \wedge \neg b\}\ c_2\ \{\psi\}}{\{\phi\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2\ \underline{\text{fi}}\ \{\psi\}}\ \text{if}$$

- $$\frac{\vdash \phi \Rightarrow \phi' \qquad \{\phi'\}\ c\ \{\psi'\} \qquad \vdash \psi' \Rightarrow \psi}{\{\phi\}\ c\ \{\psi\}}\ \text{cons}$$

Proof rule total correctness (III)

$$\dfrac{\{\phi \wedge b \wedge 0 \leq E = E_0 \}\ c\ \{\phi \wedge 0 \leq E < E_0 \}}{\{\phi \wedge 0 \leq E \}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}\ \{\phi \wedge \neg b\}}$$

where E_0 is a logical variable for retaining the initial value of E

Finding E cannot be mechanized !!!

Proof outline

- Proof outline for total correctness are similar to those for partial correctness except for
  - ☐ the precondition of the while which now writes

$$\{ \phi \wedge 0 \leq E \}$$

  - ☐ the body of the while which now writes

$$\{ \phi \wedge b \wedge 0 \leq E = E_0 \} \; c \; \{ \phi \wedge 0 \leq E < E_0 \}$$

An example

```
DIV ≡
    q := 0;
    r := x;
    while r ≥ y do
                r := r-y;
                q := q+1
    od
```

We wish to prove

$\{x \geq 0 \wedge y > 0\}$ DIV $\{ q*y+r=x \wedge 0 \leq r < y \}$

An example (II)

```
$\{x \geq 0 \wedge y > 0 \}$
$\{ 0*y+x=x \wedge 0\leq x \}$                          implied
q := 0;
$\{q*y+x=x \wedge 0\leq x \}$                           ass.
r := x;
$\{ I \}$                                               ass.
while $r \geq y$ do
        $\{ I \wedge r \geq y \}$                       Inv ∧ guard
        $\{ (q+1)*y+ r-y =x \wedge 0\leq r-y \}$        implied
        r := r-y;
        $\{ (q+1)*y+r=x \wedge 0\leq r \}$              ass.
        q := q+1
        $\{ I \}$                                       ass.
od
$\{ I \wedge r<y \}$                                    while
$\{ q*y+r=x \wedge 0\leq r<y \}$                        implied
```

where $I \equiv q*y+r=x \wedge 0\leq r$ is the invariant

An example (III)

```
$\{x \geq 0 \wedge y > 0\}$
$\{0*y+x=x \wedge 0 \leq x\}$                          implied
 q := 0;
 $\{q*y+x=x \wedge 0 \leq x\}$                         ass.
 r := x;
 $\{I \wedge 0 \leq r\}$                                ass.
 while r $\geq$ y do
     $\{I \wedge r \geq y \wedge 0 \leq r = z\}$           Inv ∧ guard
     $\{(q+1)*y + r-y = x \wedge 0 \leq r-y < z\}$     implied?????
     r := r-y;
     $\{(q+1)*y+r=x \wedge 0 \leq r < z\}$             ass.
     q := q+1
     $\{I \wedge 0 \leq r < z\}$                        ass.
 od
 $\{I \wedge r < y\}$                                   while
 $\{q*y+r=x \wedge 0 \leq r < y\}$                     implied
```

where $I \equiv q*y+r=x \wedge 0 \leq r$ is the invariant and r is the variant

An example (IV)

```
$\{x \geq 0 \wedge y > 0\}$
$\{0*y+x=x \wedge 0 \leq x \wedge y > 0\}$                implied
q := 0;
$\{q*y+x=x \wedge 0 \leq x \wedge y > 0\}$                ass.
r := x;
$\{I \wedge 0 \leq r\}$                                   ass.
while $r \geq y$ do
        $\{I \wedge r \geq y \wedge 0 \leq r = z\}$              Inv ∧ guard
        $\{(q+1)*y+r-y=x \wedge y > 0 \wedge 0 \leq r-y < z\}$    implied
        r := r-y;
        $\{(q+1)*y+r=x \wedge y > 0 \wedge 0 \leq r < z\}$       ass.
        q := q+1
        $\{I \wedge 0 \leq r < z\}$                         ass.
od
$\{I \wedge r < y\}$                                      while
$\{q*y+r=x \wedge 0 \leq r < y\}$                         implied
```

where $I \equiv q*y+r=x \wedge 0 \leq r \wedge y > 0$ is the invariant and r is the variant