Verification of Timed Systems

Wang Yi
Uppsala University

ARTIST/China spring school
XiAn, China, April 03-15, 2006

Contributors

@UPPsala
- John Håkansson
- Pavel Krcal
- Leonid Mokrushin
- Paul Pettersson
- ...

@AALborg
- Kim G Larsen
- Gerd Behrman
- Alexandre David
- ...

@Elsewhere
- **Johan Bengtsson, Fredrik Larsson**, Kåre J Kristoffersen, Tobias Amnell, Thomas Hune, Oliver Möller, Elena Fersman, Carsten Weise, David Griffioen, Ansgar Fehnker, Frits Vandraager, J-P Katoen, Martijn Hendriks, Magnus Lindahl, Justin Pearson...

OUTLINE

- A Brief Introduction
  - Motivation ... what are the problems to solve
  - CTL, LTL and basic model-checking algorithms
- Timed Systems
  - Timed automata and verification problems
  - UPPAAL tutorial (1): data stuctures & algorithms
  - UPPAAL tutorial (2): input languages
  - TIMES: From models to code ”guaranteeing” timing constraints
- Further topics/Recent Work
  - Systems with buffers/queues [CAV 2006]

Main references (Papers)

- Temporal Logics (CTL,LTL)
  - **Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach**. Edmund M. Clarke, E. Allen Emerson, A. Prasad Sistla, POPL 1983: 117-126, also as ”Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. Program. Lang. Syst. 8(2): 244-263 (1986) ”
  - **An Automata-Theoretic Approach to Automatic Program Verification,** Moshe Y. Vardi, Pierre Wolper: LICS 1986: 332-344. Also as ” Reasoning About Infinite Computations. Inf. Comput. 115(1): 1-37 (1994)”
- Timed Systems (Timed Automata, TCTL)
  - **A Theory of Timed Automata**. Rajeev Alur, David L. Dill. Theor. Comput. Sci. 126(2): 183-235 (1994)”
  - **Symbolic Model Checking for Real-Time Systems,** *Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Information and Computation* 111:193-244, 1994.
  - **UPPAAL in a Nutshell**. Kim Guldstrand Larsen, Paul Pettersson, Wang Yi. STTT 1(1-2): 134-152 (1997)
  - **Timed Automata – Semantics, Algorithms and Tools**, a tutorial on timed automata Johan Bengtsson and Wang Yi: (a book chapter in Rozenberg et al, 2004, LNCS).

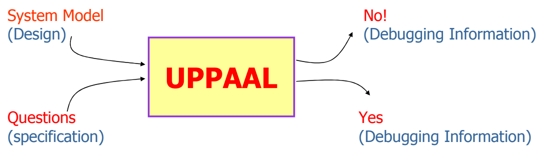
Main references (Books)

- Edmund M. Clarke, Orna Grumberg and Doron A. Peled, **Model Checking**
- G.J. Holzmann, Prentice Hall 1991, Design and Validation of Computer Protocols (new book: **The SPIN MODEL CHECKER Primer and Reference Manual** , 2003)
- Joost-Pieter Katoen, **Concepts, Algorithms, and Tools for Model Checking** (draft book on the web)

Main Goal

What’s inside the tools: UPPAAL & TIMES

UPPAAL *A model checker for real-time systems*



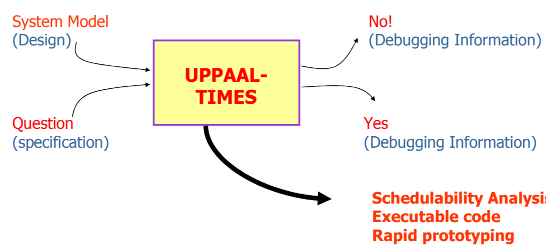
UPPAAL: www.uppaal.com

- Developed jointly by
  - Uppsala university, Sweden
  - Aalorg university, Denmark
- UPPsala + AALborg = UPPAAL
  - SWEDEN + DENMARK = SWEDEN
  - SWEDEN + DENMARK = DENMARK

TIMES: www.timestool.com

- A branch of UPPAAL, developed at Uppsala
- TIMES = a Tool for Modeling and Implemenation of Embedded Systems

TIMES *a tool for resource scheduling and code synthesis*

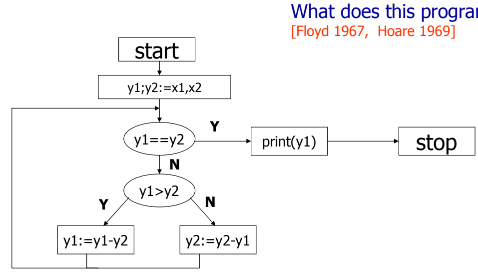


Lecture 1

Introduction

The dream started 40 years ago in 1960's aiming at "bug-free software"

What does this program do?
[Floyd 1967, Hoare 1969]



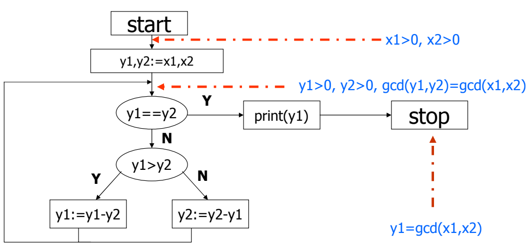
It computes the Greatest Common Divisor (gcd) of x1 and x2 [Floyd 67]

Specification (*partial correctness*)

Hoare logic: {P} program {Q} [Floyd 1967, Hoare 1969]

- Assume, initially (pre-condition)
  - x1>0, x2>0
- After each iteration of the loop (invariant)
  - y1>0, y2>0, gcd(x1,x2) = gcd(y1,y2)
- When done (post-condition)
  - y1=gcd(x1,x2)

What does this program do?



Can you check this **?**

Yes, you may prove it manually
by induction on the number of iterations.
Question: can you automate the proof ?

Software verification (now, a hot topic)

One more example (*Total correctness*)

```
Function foo(n)
begin
if n==1 then 1
        else if even(n) then foo(n/2)
                  else foo(3*n+1)
end
```

Does this program terminate for any n? (WCET?)

Reality: 10 years later (1980's)

- The majority of programs are never proven correct! what went wrong?
  - Difficult to find and prove invariants: partial correctness
  - Difficult/impossible to prove termination: total correctness
  - Difficult to write complete specifications: what I really want?
- What to do?
  - Start another research program! In 20 years, the problems will be solved, hopefully

History: Model-checking invented in 70's/80s

[Pnueli 77, Clarke et al 83, POPL83, Sifakis et al 82]

- Temporal logics/verification
  - Check the design/model: MODEL = SPEC (not the code)
  - Finite-state, non-termininating, control-intensive, less data
  - e.g. ABP ca 140 states, 1984
- BDD-based symbolic technique [Bryant 86]
  - SMV 1990 Clarke, McMillan et al, state-space 10^{20}
- On-the-fly technique [Holzman 89]
  - SPIN, COSPAN, CESAR , KRONOS, UPPAAL etc

History: Model checking for real time systems, started in the 80s/90s

- Timed automata, timed process algebras [Alur&Dill 1990]
- KRONOS, Hytech, 1993-1995, IF 2000's
- TAB 1993, UPPAAL 1995, TIMES 2002

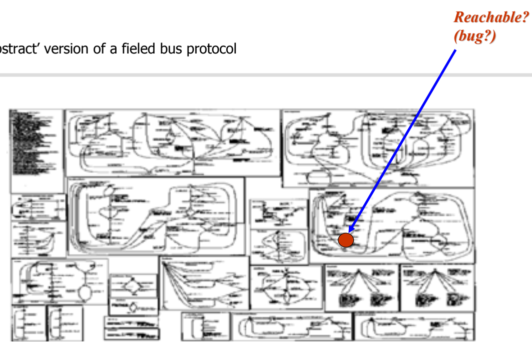
Reality: 40 years later, now

- Many extensions and improvements have been proposed, various tools exist: (non-)commercial
- Good complete specifications are still hard to obtain
- However this is not a real problem !

Reality: 40 years later, now

- Checking simple properties (e.g. deadlock freeness) is already extremely useful!
- The goal is no longer seen as proving that a system is completely, absolutely and undoutedly correct (bug-free)
- The objective is to have tools that can help a developer find errors and gain confidence in her/his design. That is achievable
- Now widely used in hardware design, protocol design, and hopefully soon, embedded systems!

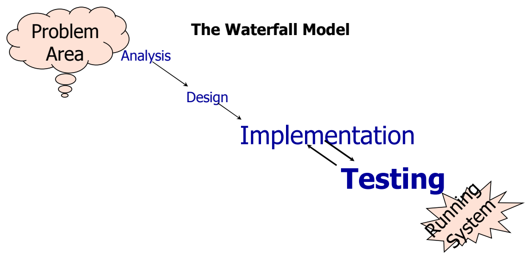
An 'abstract' version of a fieled bus protocol



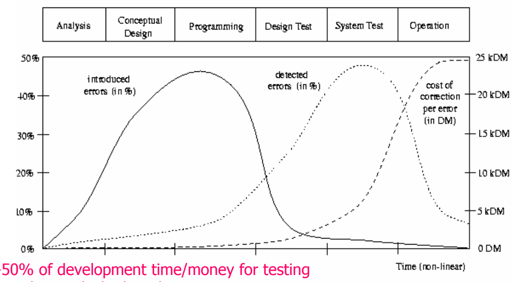
Why testing not good enough

- **Testing/simulation**: coverage problems, difficult to deal with non-determinism and concurrent computation
- **Formal verification/Model-Checking** (= **exhaustive testing** of software and hardware **design**) provides 100% coverage

Traditional software development



Introducing, Detecting and Correcting errors



- 30-50% of development time/money for testing
- Errors detected: the late the more expensive

Model-Checking may complement testing to find (design) Bugs as early as possible

Model-Checking
in a Nutshell

EXAMPLE: Petersson's algorithm

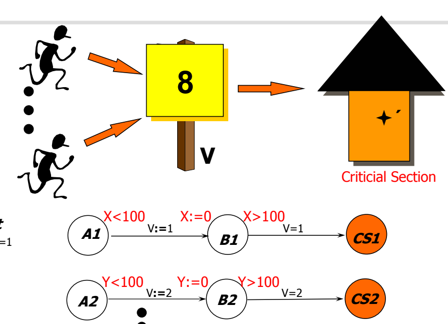
turn, flag1, flag2: shared variable

- Process 1
- loop
- flag1:=1; turn:=2
- while (flag2 & turn=2) wait
- **CS1**
- flag1:=0
- end loop

- Process 2
- loop
- flag2:=1; turn:=1
- while (flag1 & turn=1) wait
- **CS2**
- flag2:=0
- end loop

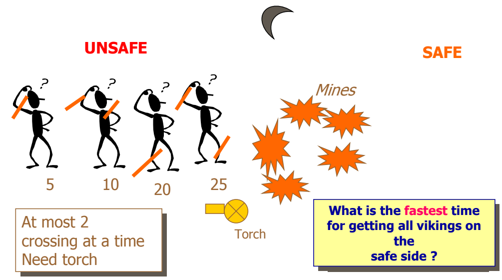
Question: can both run in CS simultaneusly ?

Example: Fischer's Protocol

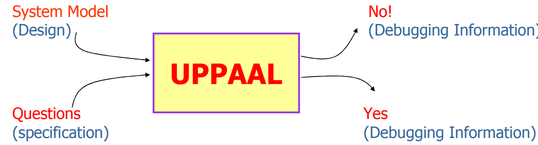


Example: the Vikings Problem

Real time scheduling



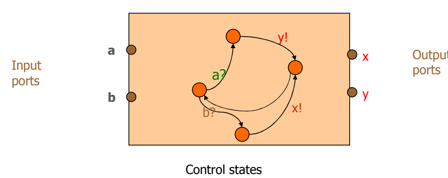
UPPAAL *A model checker for real-time systems*



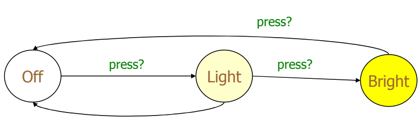
MODELING

How to construct Model ?

Program as State Machine!

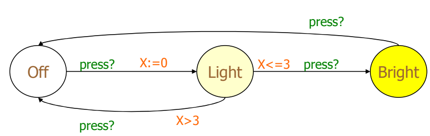


A Light Controller



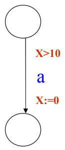
WANT: if press is issued twice quickly
then the light will get brighter; otherwise the light is
turned off.

A Light Controller (with timer)



Solution: Add real-valued clock x

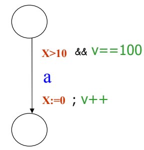
Modeling Real Time Systems



Timed Automaton

- Events
  - synchronization
  - interrupts
- Timing constraints
  - specifying event arrivals
  - e.g. Periodic and sporadic

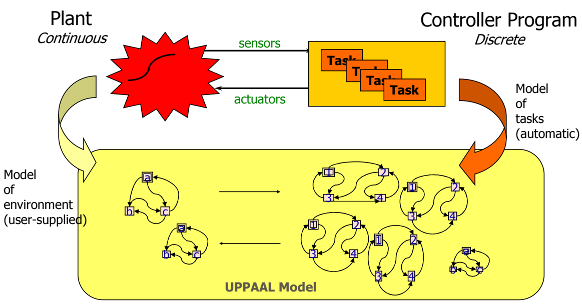
Modeling Real Time Systems

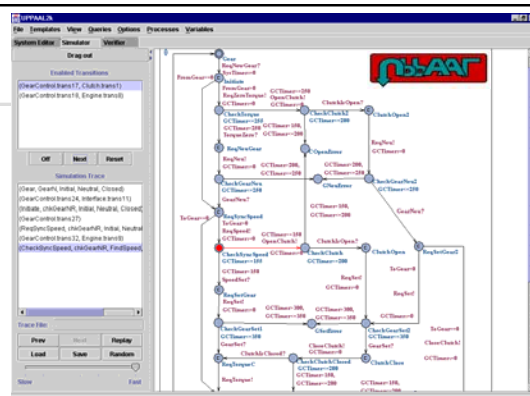


Timed Automaton in UPPAAL

- Events
  - synchronization
  - interrupts
- Timing constraints
  - specifying event arrivals
  - e.g. Periodic and sporadic
- Data variables & C-subset
  - Guards
  - assignments

Construction of Models: Concurrency





SPECIFICATION

How to ask questions: Specs ?

Specification=Requirement, Lamport 1977

- Safety
  - Something (bad) will not happen
- Liveness
  - Something (good) must happen

Specification=Requirement [Lamport 1977]

- Safety
  - Something (bad) will not happen
- Liveness
  - Something (good) must happen
- Realizability (for systems with limited resources)
  - Schedulability, enough resources?

43

Specification: Examples

- Safety
  - AG ¬(P1.CS1 & P2.CS2) **A**lways **G**lobally
  - AG (m< 100)
  - EF (5<6) Possibly in **F**uture
    - construct the whole state space
    - Report deadlocks etc.
  - EF (viking1.safe & viking2.safe & viking3.safe & viking4.safe)
  - AG (time>60 imply viking4.safe)
- Liveness
  - AF (m>100) Eventually
  - AG (P1.try imply AF P1.CS1) Leads to

44

VERIFICATION

Model meets Specs ?

45

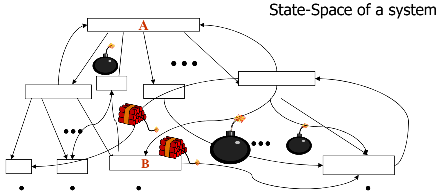
(Formal) Verification

- Semantics of a system
  = all states + state transitions
  (all possible executions)
- Verification
  = state space exploration + examination

46

Verificatioin = Searching

State-Space of a system



(1) SAFETY:
-- Is it possible to fire the bombs?
-- Is it possible to go from A to B within 10 sec?
(2) LIVENESS:
-- Will B be executed eventually (no time bound given)?

47

Approaches to Verification

- Manual: Proof systems, paper and pen
  - Find invariants (difficult !)
  - Induction: Assume nth-state OK, check (n+1)th OK
  - Boring ☹ (more fun with programming)
- Semi-automatic: Theorem proving
  - Use theorem provers to prove the induction step
  - e.g. PVS, HOL, ALF
  - Require too much expertise ☹
- Automatic: Model-Checking ☺
  - State-Space Exploration and Examination
  - e.g. SPIN, SMV, UPPAAL

48

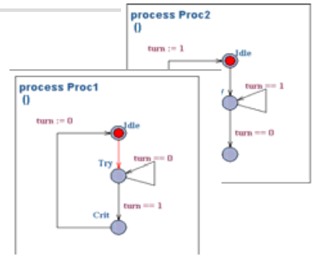
Two basic verification algorithms

- Reachability analysis
  - Checking safety properties
- Loop detection
  - Checking liveness properties

Modelling in UPPAAL: example

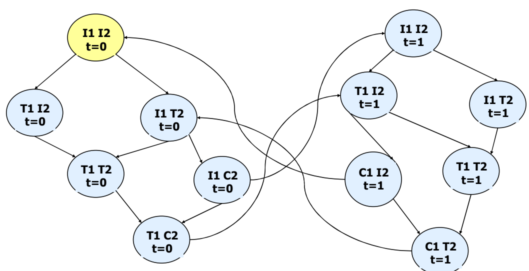
```
P1 :: while True do
      T1 : wait(turn=1)
      C1 : CS1; turn:=0
      endwhile
||
P2 :: while True do
      T2 : wait(turn=0)
      C2 : CS2; turn:=1
      endwhile
```

Mutual Exclusion Program



Is it possible that P1 and P2 run C1 and C2 simultaneously?

Verification: example

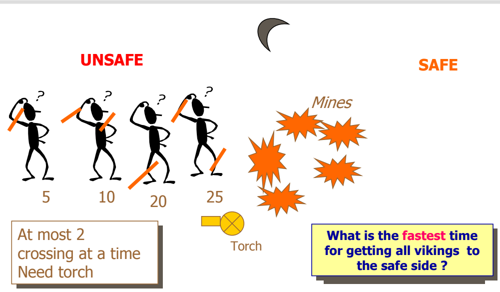


(C1,C2) is not reachable!

UPPAAL Demo

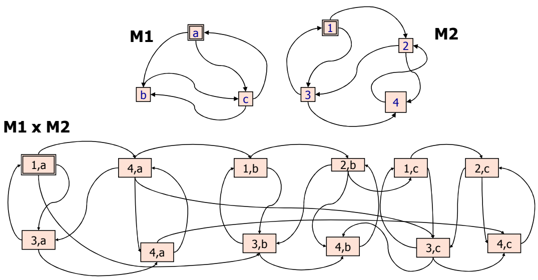
Example: the Vikings Problem

Real time scheduling



This sounds too good!
What's the problem?

Problem with verification: 'State Explosion'



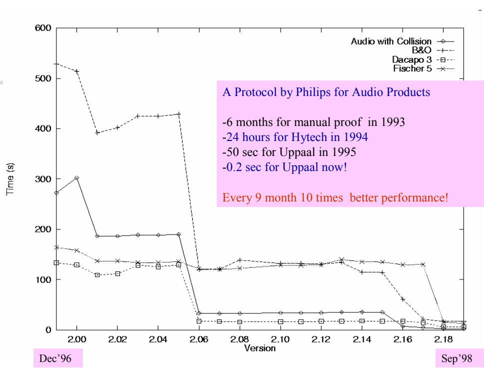
All combinations = exponential in no. of components

EXAMPLE

13 components and each with 1 clock & 10 states

# of states = 10,000,000,000,000 =10,000 G
Each needs (10 * 10)* 4Bytes = 400 Bytes

Worst case memory usage >> 4,000,000GB



The dream goes on … …

- *Model Checking, a useful and applicable technique as compiler theory*

End of introduction

OUTLINE

- A Brief Introduction
  - Motivation … what are the problems to solve
  - CTL, LTL and basic model-checking algorithms
- Timed Systems
  - Timed automata and verification problems
  - UPPAAL tutorial (1): data stuctures & algorithms
  - UPPAAL tutorial (2): input languages
  - TIMES: From models to code "guaranteeing" timing constraints
- Further topics/Recent Work
  - Systems with buffers/queues [CAV 2006]

Lecture 2

Model-Checking Untimed Systems

Transition Systems, Temporal Logics and
Basic Verification Algorithms

Transition Systems

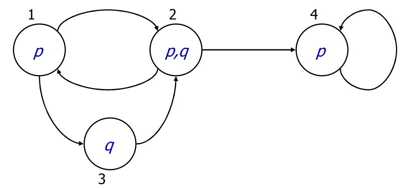
A transition system is a graph with

- a set of nodes (states)
- a set of edges (transitions)

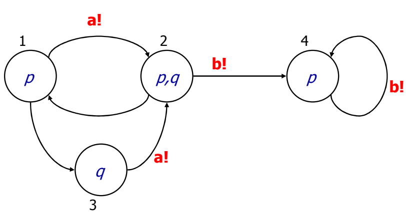
where

- nodes may be labeled with propositions (state properties)
- edges may be labeled with action names (synchronization)

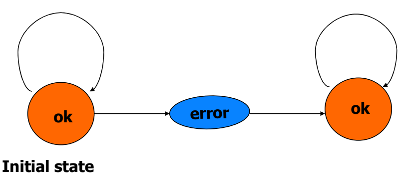
Example (with labeled nodes)



EXAMPLE (with labeled nodes, and labeled edges)

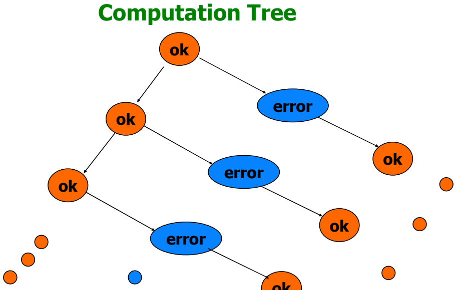


EXAMPLE: a BUGGY machine



The (branching-time) semantic of BUGGY

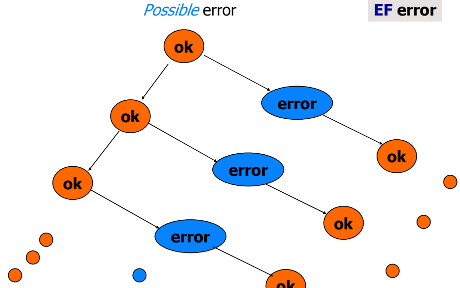
Computation Tree



"Properties" of BUGGY

Possible error

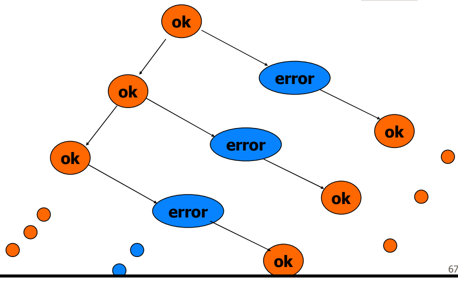
EF error



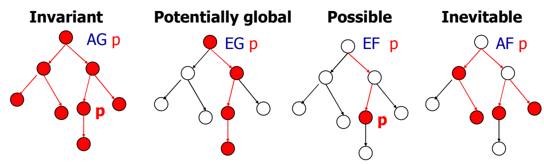
"Properties" of BUGGY

Possoble Globally ok

EG ok



Properties of Computation Trees



CTL: Computation Tree Logics

defined on Computation Trees of Kripke structures

Computation Tree Logic, CTL

Clarke & Emerson 1980

Syntax

ϕ **::=** **P** | $\neg\phi$ | $\phi \vee \phi$ | **EX** ϕ | **E[**ϕ **U** ϕ**]** | **A[**ϕ **U** ϕ**]**

where **P** $\in$ AP (atomic propositions)

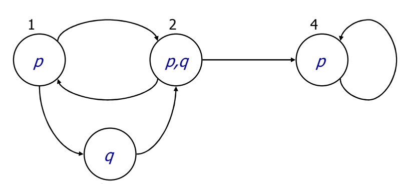
- EX (pronounced "for some path next")
- E (pronounced "for some path")
- A (pronounced "for all paths") and
- U (pronounced "until").

A CTL-model is a Kripke Structure (=transition systems with labeled nodes)

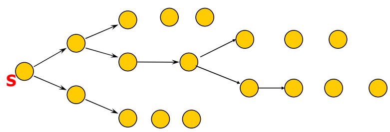
M = <S, E, Label> where

- S is a non-empty set of states
- $E \subseteq S \times S$ is a transition relation
- Label: $S \rightarrow 2^{AP}$ is a labeling function

Example

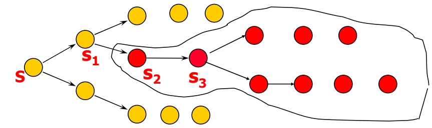


Computation Trees vs. STATES



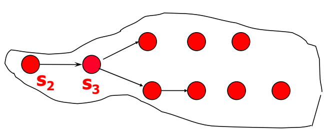
The computation tree of state s

Computation Trees vs. STATES



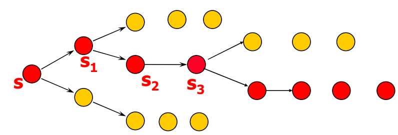
The computation tree of state s_2

Computation trees of STATES



The computation tree of state s_2

Path (of computation tree)



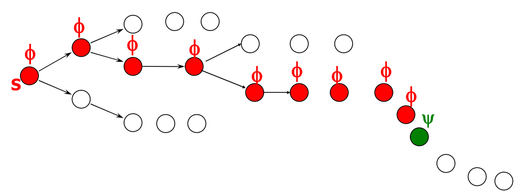
A path is an infinite sequence of states
e.g. $\sigma = s \ s_1 \, s_2 \, s_3 \ \ldots \ldots$

Formal Semantics of CTL

$s \models p$	iff $p \in Label(s)$
$s \models \neg\phi$	iff $\neg(s \models \phi)$
$s \models \phi \vee \psi$	iff $(s \models \phi) \vee (s \models \psi)$
$s \models \mathsf{EX}\,\phi$	iff $\exists \sigma \in P_{\mathcal{M}}(s).\, \sigma[1] \models \phi$
$s \models \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$	iff $\exists \sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\, \sigma[j] \models \psi \ \wedge \ (\forall 0 \leqslant k < j.\, \sigma[k] \models \phi))$
$s \models \mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$	iff $\forall \sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\, \sigma[j] \models \psi \ \wedge \ (\forall 0 \leqslant k < j.\, \sigma[k] \models \phi))$.

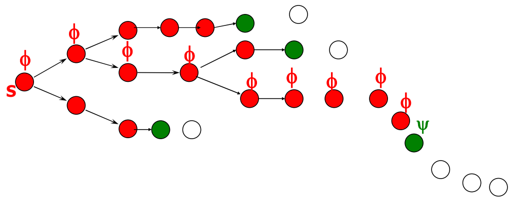
Where $P_M(s)$ denotes the set of pathes starting from s and $\sigma[i]$ denotes i'th element of σ

E[ϕ U ψ]



E[ϕ U ψ] is valid in **s** if <u>some path</u> from **s** satisfies the above

A[ϕ U ψ]

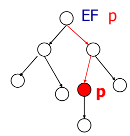


A[ϕ U ψ] is valid in **s** if <u>all pathes</u> from **s** satisfy the above

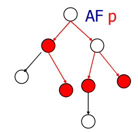
CTL, Derived Operators

$$\mathrm{EF}\,\phi \;\equiv\; \mathrm{E}\,[\mathrm{true}\;\mathrm{U}\;\phi]$$ *possible*

$$\mathrm{AF}\,\phi \;\equiv\; \mathrm{A}\,[\mathrm{true}\;\mathrm{U}\;\phi].$$ *inevitable*



E<> P in UPPAAL

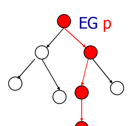


A<> P in UPPAAL

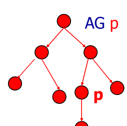
CTL, Derived Operators

$$\mathrm{EG}\,\phi \;\equiv\; \neg\,\mathrm{AF}\,\neg\,\phi$$ *potentially always*

$$\mathrm{AG}\,\phi \;\equiv\; \neg\,\mathrm{EF}\,\neg\,\phi$$ *always*

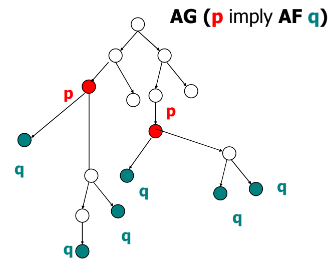


E[] P in UPPAAL



A[] P in UPPAAL

CTL, Derived Operators (cont.)



p - -> q in **UPPAAL**

Theorem

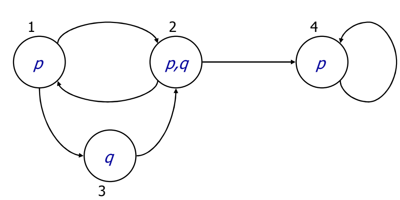
All operators are derivable from

- EX f
- EG f
- E[f U g]

and boolean connectives

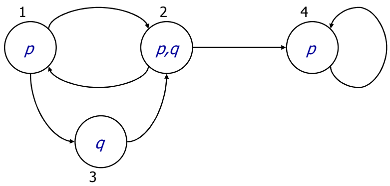
$$\mathrm{A}[f\;\mathrm{U}\;g] \equiv \neg\mathrm{E}[\neg g\,\mathrm{U}(\neg f \wedge \neg g)] \wedge \neg\mathrm{EG}\neg g$$

Example



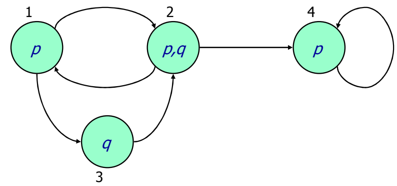
Example

EX *p*



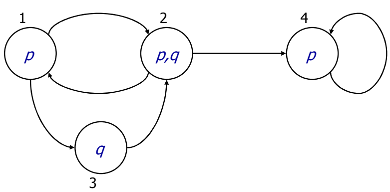
Example

EX *p*



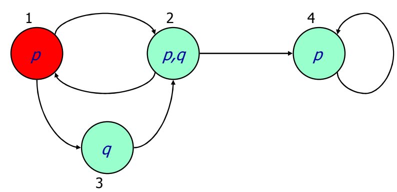
Example

AX *p*



Example

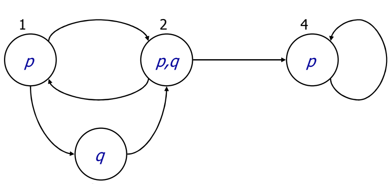
AX *p*



Note: state 1 doesn't satisfy AX *p*

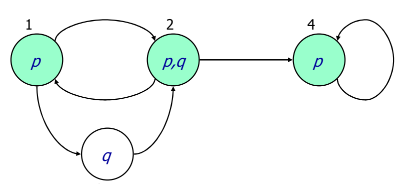
Example

EG *p*



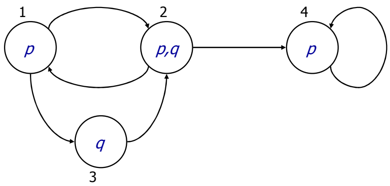
Example

EG *p*



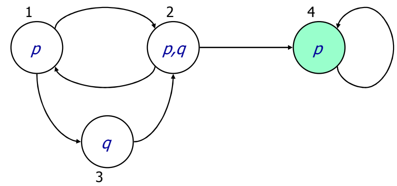
Example

AG p



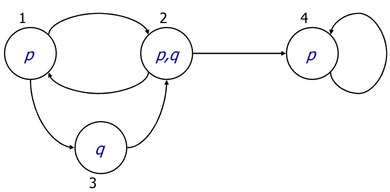
Example

AG p



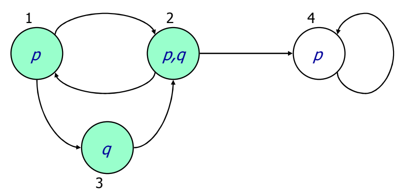
Example

A[p **U** q **]**



Example

A[p **U** q **]**



Properties of MUTEX example ?

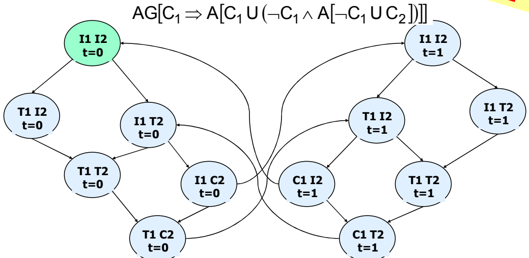
$AG\ \neg(C_1 \wedge C_2)$

$AG[\ T_1 \Rightarrow AF(C_1)]$

$EG[\neg C_1]$

$AG[C_1 \Rightarrow A[C_1\ U\ (\neg C_1 \wedge A[\neg C_1\ U\ C_2])]]$

HOW to DECIDE IN GENERAL



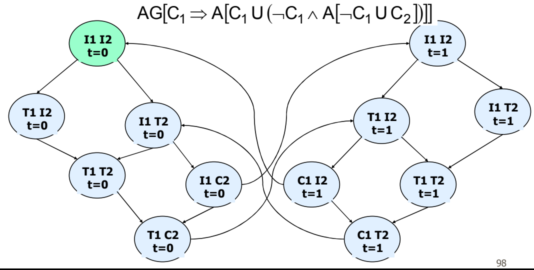
CTL Model-Checking Algorithms

Labeling Methods [Clarke et al 81]

- $Sat(\phi)$ = all states where ϕ is true
- Compute $Sat(\phi)$ recursively as follows:
  - For each sub-formula ϕ_i of ϕ, compute $Sat(\phi_i)$
  - This is easier: e.g. $Sat(P) = \{s \mid P \in Label(s)\}$
- Compose $Sat(\phi_i)$ to get $Sat(\phi)$

Properties of MUTEX example ?

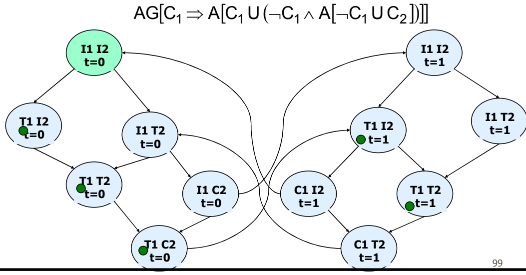
$AG\,\neg(C_1 \wedge C_2)$
$AG[\,T_1 \Rightarrow AF(C_1)]$
$EG[\neg C_1]$
$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



Properties of MUTEX example ?

Compute Sat(T_1)

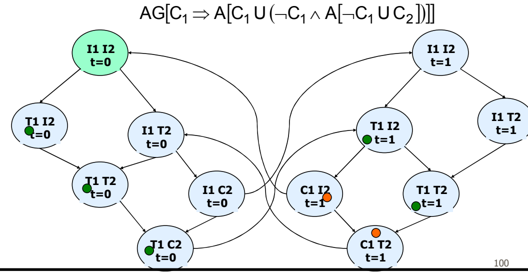
$AG\,\neg(C_1 \wedge C_2)$
$AG[\,T_1 \Rightarrow AF(C_1)]$
$EG[\neg C_1]$
$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



Properties of MUTEX example ?

Compute Sat(C_1)

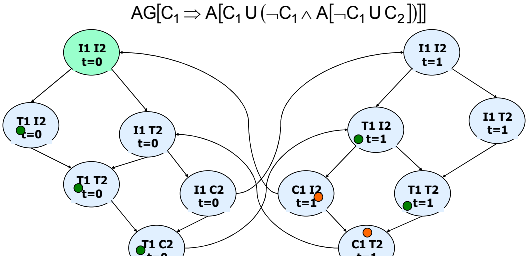
$AG\,\neg(C_1 \wedge C_2)$
$AG[\,T_1 \Rightarrow AF(C_1)]$
$EG[\neg C_1]$
$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



Properties of MUTEX example ?

Compute Sat(AF C_1)

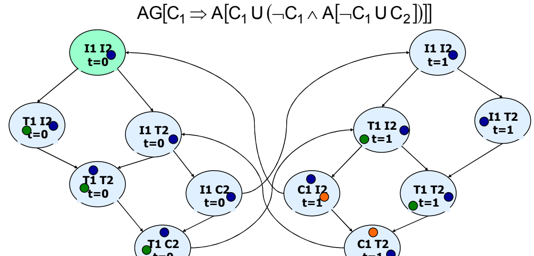
$AG\,\neg(C_1 \wedge C_2)$
$AG[\,T_1 \Rightarrow AF(C_1)]$
$EG[\neg C_1]$
$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



Properties of MUTEX example ?

Compute Sat(AF C_1)

$AG\,\neg(C_1 \wedge C_2)$
$AG[\,T_1 \Rightarrow AF(C_1)]$
$EG[\neg C_1]$
$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



Properties of MUTEX example ?

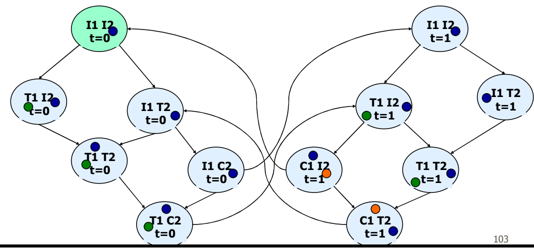
Sat(T_1 => AF C_1) ?

$AG\,\neg(C_1 \wedge C_2)$

$AG[\,T_1 \Rightarrow AF(C_1)]$

$EG[\neg C_1]$

$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



Properties of MUTEX example ?

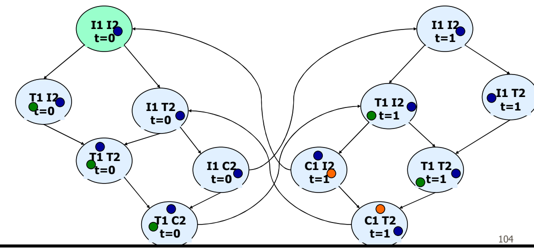
Sat(T_1 => AF C_1) = all states !!

$AG\,\neg(C_1 \wedge C_2)$

$AG[\,T_1 \Rightarrow AF(C_1)]$

$EG[\neg C_1]$

$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



Properties of MUTEX example ?

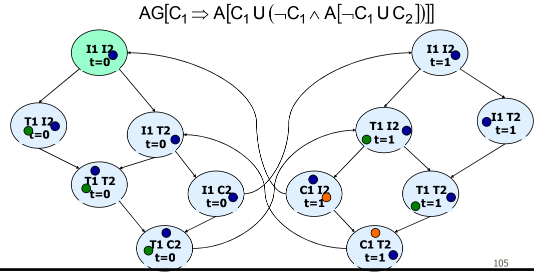
Sat(AG[T_1 => AF C_1]) = all states !!

$AG\,\neg(C_1 \wedge C_2)$

$AG[\,T_1 \Rightarrow AF(C_1)]$

$EG[\neg C_1]$

$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$



function $Sat\,(\phi : Formula)$: **set of** $State$;
(* precondition: true *)
begin
- **if** $\phi = \text{true} \longrightarrow$ **return** S
- $[]\ \phi = \text{false} \longrightarrow$ **return** $\varnothing$
- $[]\ \phi \in AP \longrightarrow$ **return** $\{\, s \mid \phi \in Label(s)\,\}$
- $[]\ \phi = \neg\,\phi_1 \longrightarrow$ **return** $S - Sat(\phi_1)$
- $[]\ \phi = \phi_1 \vee \phi_2 \longrightarrow$ **return** $(Sat(\phi_1) \cup Sat(\phi_2))$
- $[]\ \phi = \mathsf{EX}\,\phi_1 \longrightarrow$ **return** $\{\, s \in S \mid (s,s') \in R \;\wedge\; s' \in Sat(\phi_1)\,\}$
- $[]\ \phi = \mathsf{E}\,[\phi_1\,\mathsf{U}\,\phi_2] \longrightarrow$ **return** $Sat_{EU}(\phi_1,\phi_2)$
- $[]\ \phi = \mathsf{A}\,[\phi_1\,\mathsf{U}\,\phi_2] \longrightarrow$ **return** $Sat_{AU}(\phi_1,\phi_2)$
- **fi**

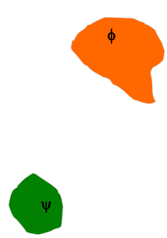
(* postcondition: $Sat(\phi) = \{\, s \mid \mathcal{M}, s \models \phi\,\}$ *)
end

How to Compute Sat(E[ϕ U ψ])

function $Sat_{EU}(\phi,\psi : Formula)$: set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
  $Q, Q' := Sat(\psi), \varnothing$;
  do $Q \neq Q' \longrightarrow$
    $Q' := Q$;
    $Q := Q \cup (\{\, s \mid \exists s' \in Q.\,(s,s') \in R\,\} \cap Sat(\phi))$
  od;
  return Q
(* postcondition: $Sat_{EU}(\phi,\psi) = \{\, s \in S \mid \mathcal{M}, s \models \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]\,\}$ *)
end

Table 3.4: Labelling procedure for $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$

Passed

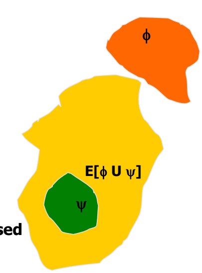


How to Compute Sat(E[ϕ U ψ])

function $Sat_{EU}(\phi,\psi : Formula)$: set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
  $Q, Q' := Sat(\psi), \varnothing$;
  do $Q \neq Q' \longrightarrow$
    $Q' := Q$;
    $Q := Q \cup (\{\, s \mid \exists s' \in Q.\,(s,s') \in R\,\} \cap Sat(\phi))$
  od;
  return Q
(* postcondition: $Sat_{EU}(\phi,\psi) = \{\, s \in S \mid \mathcal{M}, s \models \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]\,\}$ *)
end

Table 3.4: Labelling procedure for $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$

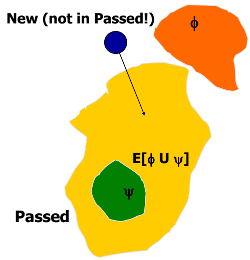
Passed



How to Compute Sat(E[ϕ U ψ])

```
function Sat_EU(φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
    Q, Q' := Sat(ψ), ∅;
    do Q ≠ Q' →
        Q' := Q;
        Q := Q ∪ ({s | ∃s' ∈ Q. (s, s') ∈ R} ∩ Sat(φ))
    od;
    return Q
(* postcondition: Sat_EU(φ, ψ) = {s ∈ S | M, s ⊨ E[φ U ψ]} *)
end
```

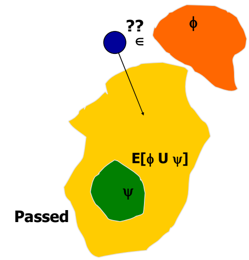
Table 3.4: Labelling procedure for E[φ U ψ]



How to Compute Sat(E[ϕ U ψ])

```
function Sat_EU(φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
    Q, Q' := Sat(ψ), ∅;
    do Q ≠ Q' →
        Q' := Q;
        Q := Q ∪ ({s | ∃s' ∈ Q. (s, s') ∈ R} ∩ Sat(φ))
    od;
    return Q
(* postcondition: Sat_EU(φ, ψ) = {s ∈ S | M, s ⊨ E[φ U ψ]} *)
end
```

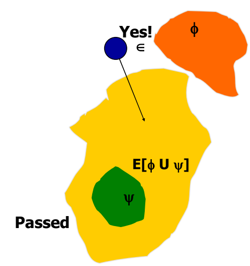
Table 3.4: Labelling procedure for E[φ U ψ]



How to Compute Sat(E[ϕ U ψ])

```
function Sat_EU(φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
    Q, Q' := Sat(ψ), ∅;
    do Q ≠ Q' →
        Q' := Q;
        Q := Q ∪ ({s | ∃s' ∈ Q. (s, s') ∈ R} ∩ Sat(φ))
    od;
    return Q
(* postcondition: Sat_EU(φ, ψ) = {s ∈ S | M, s ⊨ E[φ U ψ]} *)
end
```

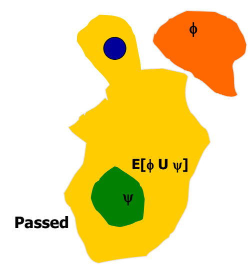
Table 3.4: Labelling procedure for E[φ U ψ]



How to Compute Sat(E[ϕ U ψ])

```
function Sat_EU(φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
    Q, Q' := Sat(ψ), ∅;
    do Q ≠ Q' →
        Q' := Q;
        Q := Q ∪ ({s | ∃s' ∈ Q. (s, s') ∈ R} ∩ Sat(φ))
    od;
    return Q
(* postcondition: Sat_EU(φ, ψ) = {s ∈ S | M, s ⊨ E[φ U ψ]} *)
end
```

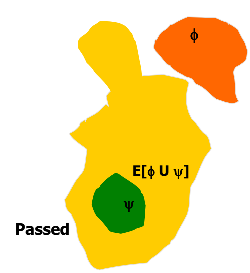
Table 3.4: Labelling procedure for E[φ U ψ]



How to Compute Sat(E[ϕ U ψ])

```
function Sat_EU(φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
    Q, Q' := Sat(ψ), ∅;
    do Q ≠ Q' →
        Q' := Q;
        Q := Q ∪ ({s | ∃s' ∈ Q. (s, s') ∈ R} ∩ Sat(φ))
    od;
    return Q
(* postcondition: Sat_EU(φ, ψ) = {s ∈ S | M, s ⊨ E[φ U ψ]} *)
end
```

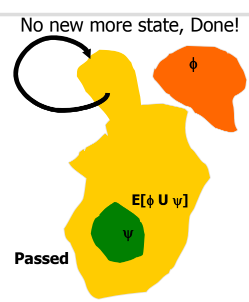
Table 3.4: Labelling procedure for E[φ U ψ]



How to Compute Sat(E[ϕ U ψ])

```
function Sat_EU(φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
    Q, Q' := Sat(ψ), ∅;
    do Q ≠ Q' →
        Q' := Q;
        Q := Q ∪ ({s | ∃s' ∈ Q. (s, s') ∈ R} ∩ Sat(φ))
    od;
    return Q
(* postcondition: Sat_EU(φ, ψ) = {s ∈ S | M, s ⊨ E[φ U ψ]} *)
end
```

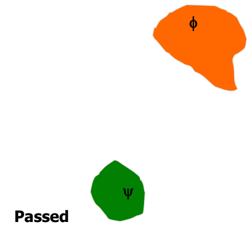
Table 3.4: Labelling procedure for E[φ U ψ]



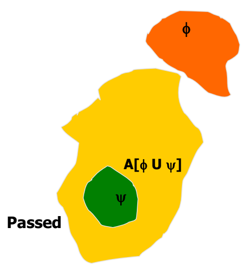
How to Compute Sat(A[ϕ U ψ])

?

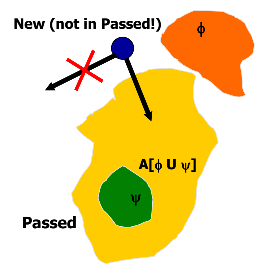
How to Compute Sat(A[ϕ U ψ])



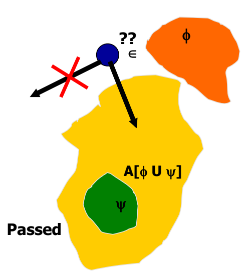
How to Compute Sat(A[ϕ U ψ])



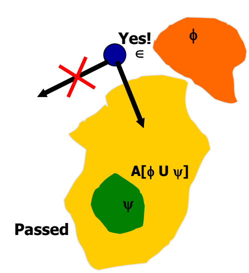
How to Compute Sat(A[ϕ U ψ])



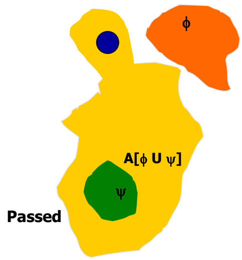
How to Compute Sat(A[ϕ U ψ])



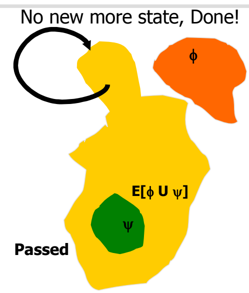
How to Compute Sat(A[ϕ U ψ])



How to Compute Sat(A[ϕ U ψ])



How to Compute Sat(A[ϕ U ψ])



function $Sat_{AU}\,(\phi,\psi : Formula)$: **set of** $State$;
(* precondition: true *)
begin var Q, Q' : **set of** $State$;
$\quad Q, Q' := Sat(\psi), \varnothing$;
$\quad$ **do** $Q \neq Q' \longrightarrow$
$\qquad Q' := Q$;
$\qquad Q := Q \cup (\{s \mid \forall s'.(s,s') \in R \Rightarrow s' \in Q\} \cap Sat(\phi))$
$\quad$ **od**;
$\quad$ **return** Q
(* postcondition: $Sat_{AU}(\phi,\psi) = \{ s \in S \mid \mathcal{M}, s \models \mathsf{A}\,[\phi \,\mathsf{U}\, \psi] \}$ *)
end

Table 3.5: Labelling procedure for $\mathsf{A}\,[\phi \,\mathsf{U}\, \psi]$

Fixpoint Characterizations (SMV)

$$\mathrm{EF}\, p \;\equiv\; p \vee \mathrm{EXEF}\, p$$

Let A be the set of states satisfying EF p then

$$\mathrm{A} \;\equiv\; p \vee \mathrm{EX\,A}$$

in fact A is the smallest one of sets satisfying the equations (the least fixpoint)

Fixed points of monotonic functions

- Let τ be a function $S \rightarrow S$
- Say τ is *monotonic* when
$$x \subseteq y \text{ implies } \tau(x) \subseteq \tau(y)$$
- Fixed point of τ is y such that
$$\tau(y) = y$$
- If τ monotonic, then it has
  - least fixed point $\mu y.\ \tau(y)$
  - greatest fixed point $\nu y.\ \tau(y)$

Iteratively computing fixed points

- Suppose S is finite
  - The least fixed point $\mu y.\ \tau(y)$ is the limit of
$$\text{false} \;\subseteq\; \tau(\text{false}) \;\subseteq\; \tau(\tau(\text{false})) \;\subseteq \Lambda$$
  - The greatest fixed point $\nu y.\ \tau(y)$ is the limit of
$$\text{true} \;\supseteq\; \tau(\text{true}) \;\supseteq\; \tau(\tau(\text{true})) \;\supseteq \Lambda$$

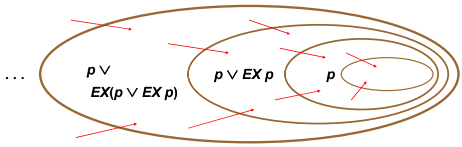
Note, since S is finite, convergence is finite

Example: *EF p*

- *EF p* is characterized by

$$EF\ p = \mu y.(p \vee EX\ y)$$

- Thus, it is the limit of the increasing series...

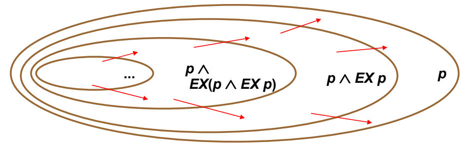


Example: *EG p*

- *EG p* is characterized by

$$EG\ p = \nu y.(p \wedge EX\ y)$$

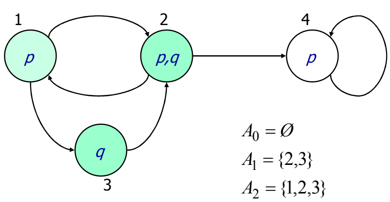
- Thus, it is the limit of the decreasing series...



Example, continued

EF *q*

$$EF\ q = \mu y.(q \vee EX\ y)$$



$A_0 = \emptyset$

$A_1 = \{2,3\}$

$A_2 = \{1,2,3\}$

$A_3 = \{1,2,3\}$

Remaining operators

$$
\begin{aligned}
AF\ p &= \mu y.(p \vee AX\ y) \\
AG\ p &= \nu y.(p \wedge AX\ y) \\
E(p\, U\, q) &= \mu y.(q \vee (p \wedge EX\ y)) \\
A(p\, U\, q) &= \mu y.(q \vee (p \wedge AX\ y))
\end{aligned}
$$

Complexity

The worst-case time complexity of checking whether system-model sys satisfies the CTL*-formula* ϕ *is* $\mathcal{O}(|S_{sys}|^2 \times |\phi|)$

However S_{sys} **may be EXPONENTIAL in number of parallel components!**
--
FIXPOINT COMPUTATIONS may be carried out using
ROBDD's
(Reduced Ordered Binary Decision Diagrams)
Bryant, 86

LTL: Linear Time Logics

defined on infinite traces (of transition systems with Buchi accepting conditions)

LTL, Linear-Time Logic

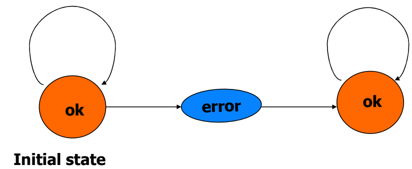
Syntax

ϕ **::= P |** $\neg \phi$ **|** $\phi \vee \phi$ **| EX** ϕ **|** ϕ **U** ϕ

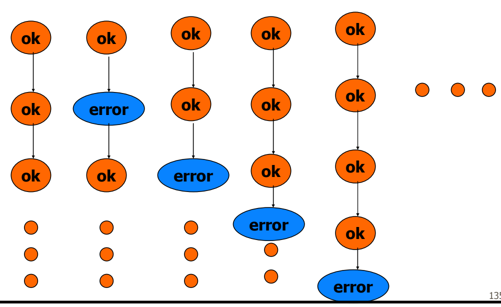
where **P** $\in$ AP (atomic propositions)

EX pronounced "n**EX**t state"
U pronounced "**U**ntil"

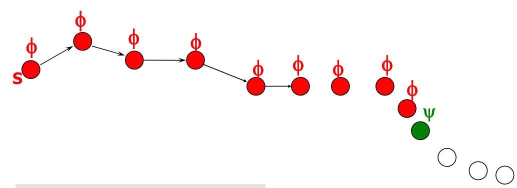
EXAMPLE: a BUGGY machine



The linear-time behaviour of BUGGY

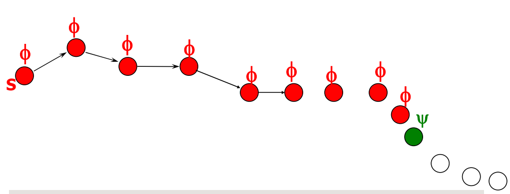


ϕ U ψ satisfied by a trace



The above trace satisfies ϕ U ψ

ϕ U ψ satisfied by a system (def.)



If all traces from initial state S of the system satisfies ϕ U ψ

Derived Operators

- $\Diamond\phi$ denotes (true U ϕ) *inevitablly*
- $\Box\phi$ denotes $\neg(\Diamond \neg \phi)$ *invariantly*/globally

Comparing CTL and LTL

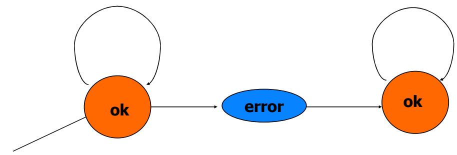
◊P (LTL) similar to AF p (CTL)
□p (LTL) similar to AG p (CTL)

However,

- LTL cannot express reachability properties: EF P in CTL
- CTL cannot express ◊□ **p** in LTL

- **CTL* = LTL + CTL**

Comparing CTL and LTL (contn.)

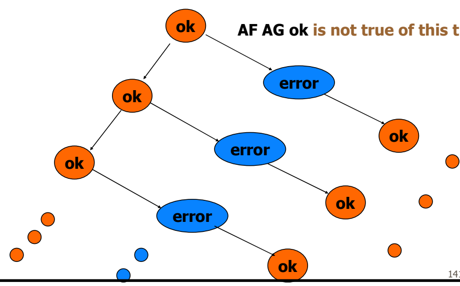


Satisfies ◊□ ok
but it does not satisfy AF AG ok

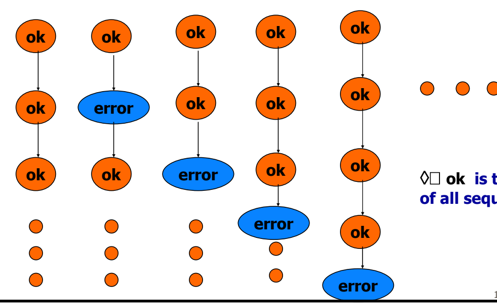
Why?

No subtree where ok is true everywhere

AF AG ok is not true of this tree



The linear-time behaviour of BUGGY



◊□ ok is true of all sequences!

Model Checking LTL [Wolper et al 1986]

- Given an automata M and a formula ϕ, to check M sat ϕ
  - Construct the formula automaton: A($\neg\phi$)
  - Construct the product automaton M || A($\neg\phi$) (on-the-fly)
- If M || A($\neg\phi$) is empty then M sat ϕ otherwise NO
- Time-Complexity = $|M| * 2^{O(|\phi|)}$

The same idea can be used for CTL model checking using Tree-automata

END
(of Untimed Systems)