Modeling & Analysis of Timed Systems

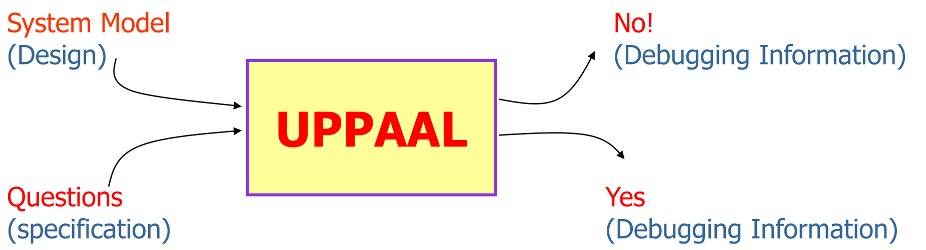
Wang Yi
Uppsala University

CUGS May 7-8, 2012

Main goal of this course

What's inside the tools: UPPAAL & TIMES
(and also some recent work on multicore timing analysis if time allows)

UPPAAL *A model checker for real-time systems*



3

UPPAAL: www.uppaal.com

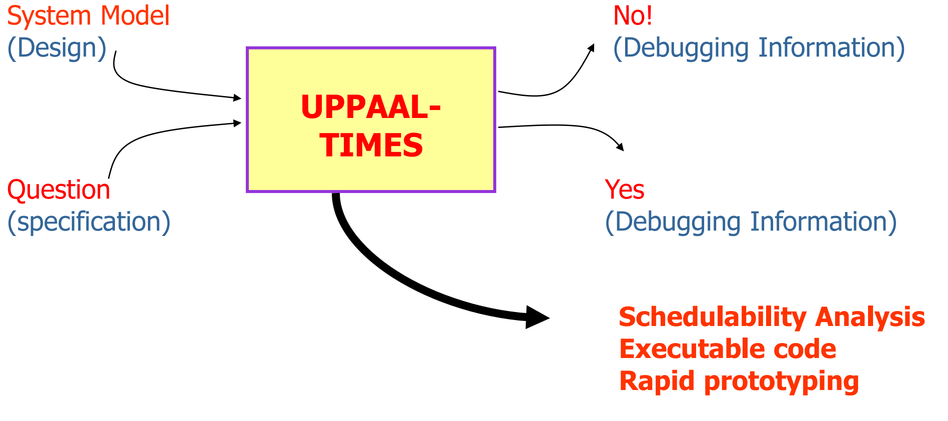
- Developed jointly by
  - Uppsala university, Sweden
  - Aalorg university, Denmark
- UPPsala + AALborg = UPPAAL
  - SWEDEN + DENMARK = SWEDEN
  - SWEDEN + DENMARK = DENMARK

4

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*



OUTLINE

- A Brief Introduction
  - Motivation ... what are the problems to solve
  - CTL, LTL and basic model-checking algorithms
- Timed Systems
  - Timed automata, TCTL and verification problems
  - UPPAAL tutorial: data stuctures & algorithms
  - TIMES: schedulability analysis using timed automata
- Recent Work
  - The multicore timing analysis problems
  - Some solutions: WCET analysis and multiprocessor scheduling

Main references (papers)

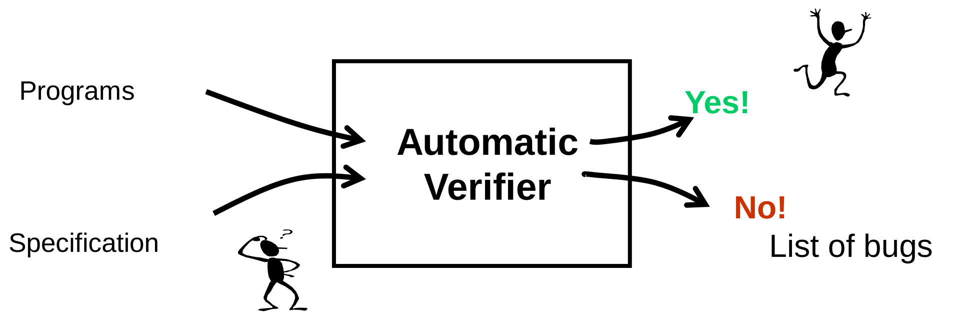
- Temporal Logics (CTL,LTL)
  - **Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications**: **A Practical Approach**. Edmund M. Clarke, E. Allen Emerson, A. Prasad Sistla, POPL 1983: 117-126, also as "Automatic Verification of Finite-State Concurrent Systems Using 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).
  - **On-line help of UPPAAL**: www.uppaal.com

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 (newer book: **The SPIN MODEL CHECKER Primer and Reference Manual** , 2003)
- Joost-Pieter Katoen and Christel Baier, **Concepts, Algorithms, and Tools for Model Checking** (MIT press)

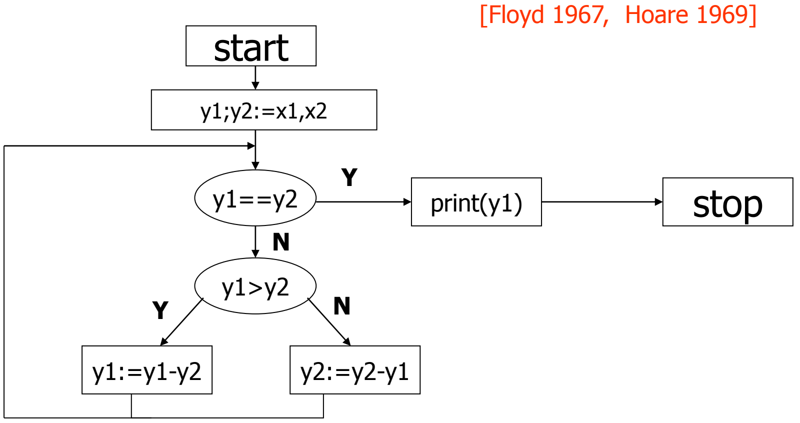
Lecture 1
Motivation and some historical remarks

Dream: Program verifier



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

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



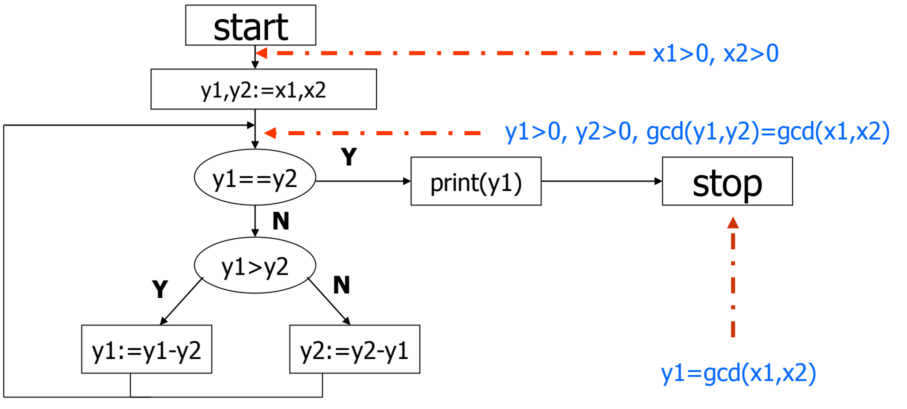
It computes the Greatest Common Divisor (gcd) of x1 and x2 [Floyd 67]

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]

- Restrict attention to finite-state programs
  - Control skeleton + boolean (finite-domain) variables
  - Found in hardware design, communication protocols, process control
- Temporal logic specification of e.g., synchronization pattern
  - There are algorithms to check that MODEL of program satisfies: SPEC
  - e.g. Alternating Bit Protocol skeleton, around 140 states, 1984
- BDD-based symbolic technique [Bryant 86]
  - SMV 1990 Clarke, McMillan et al, state-space 10^{20}
  - Now powerful tools used in processor design
- On-the-fly enumerative technique [Holzman 89]
  - SPIN, COSPAN, CAESAR, KRONOS, IF/BIP, UPPAAL etc
- SAT-based techniques [Clarke et al, McMillan, ...]

19

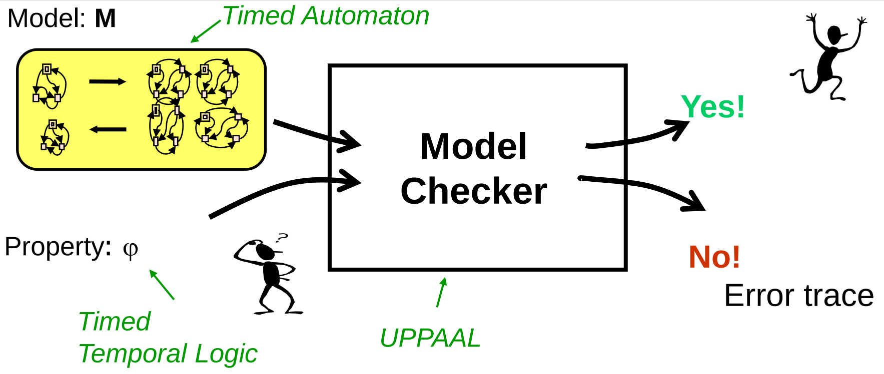
History: Model checking for real time systems, started in the 80s/90s

- Extension of model checking to consider time quantities
  - Models, specfications, and algorithms can be extended
- Timed automata, timed process algebras

  [Alur&Dill 1990]
- Tools
  - KRONOS, Hytech, 1993-1995, IF 2000's
  - TAB 1993, UPPAAL 1995, TIMES 2002

20

Model Checking



Problems that can be addresed by Model Checking

Checking correctness of

- Communication protocols
- Distributed Algorithms
- Controllers
- Hardware circuits
- Parallel and distributed software
- Embedded and real-time systems and software

e.g., Absence of race conditions, proper synchronization, ….

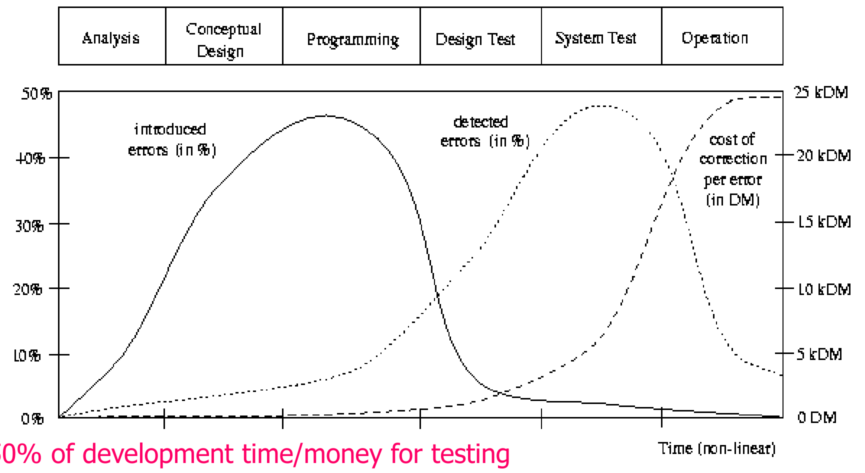
Model checking is the appropriate technique when there are many different scenarios of interaction between components in a system

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

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

Introducing, Detecting and Correcting errors



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

25

Motivation: Model Verification

Requirements

High level design

Detailed design

Build model of the design.
Analyze it thoroughly

coding

testing

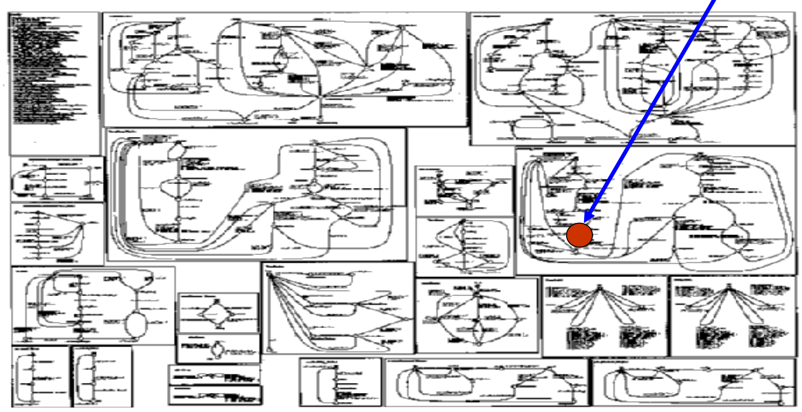
deployment

Testing concentrates more on low-level issues
And conformance to model

26

Reachable?
(bug?)

An 'abstract' version of a fieled bus protocol



Model-Checking
in a Nutshell

EXAMPLE: Petersson's algorithm

turn, flag1, flag2: shared variable

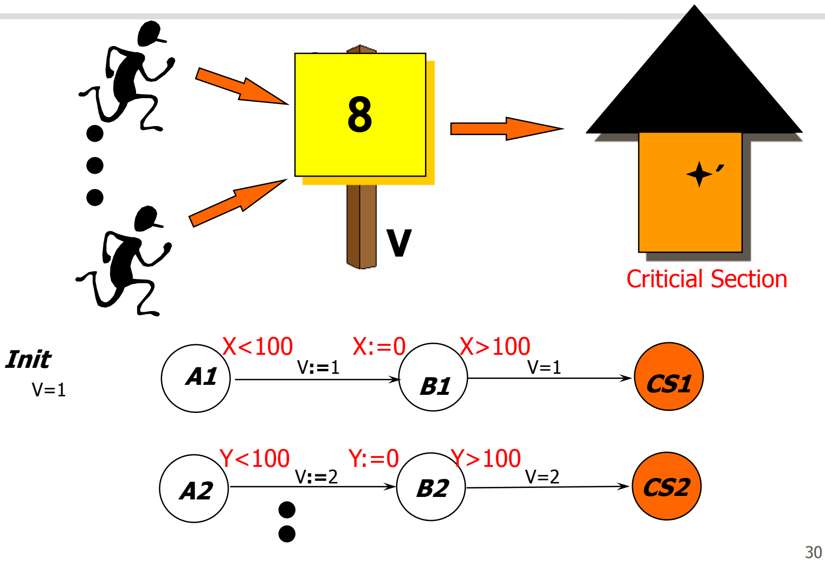
- Process 1
- loop
- flag1:=1; turn:=2
- while (flag2 & turn=2) wait
- **CS1**
- flag1:=0
- end loop

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

Question: can both run in CS simultaneusly ?

29

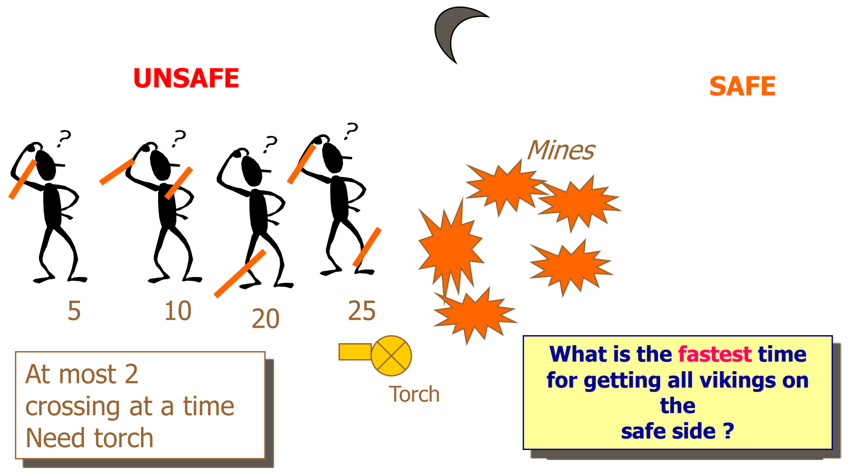
Example: Fischer's Protocol



30

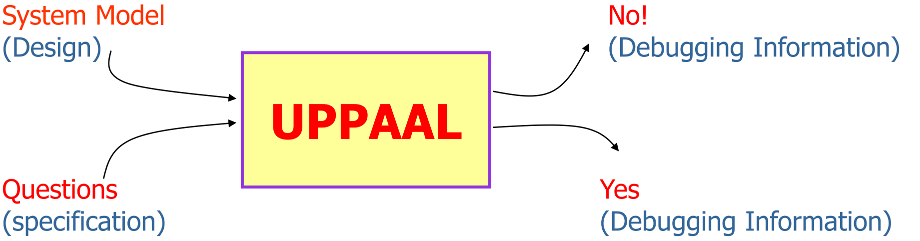
Example: the Vikings Problem

Real time scheduling



31

UPPAAL *A model checker for real-time systems*

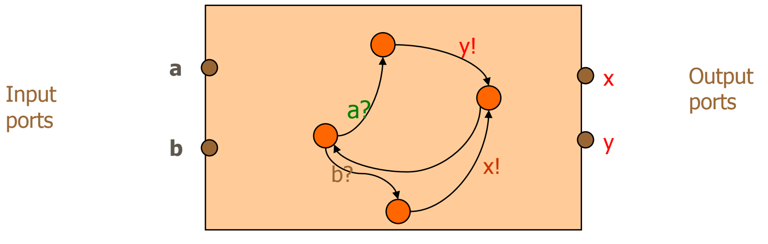


32

MODELING

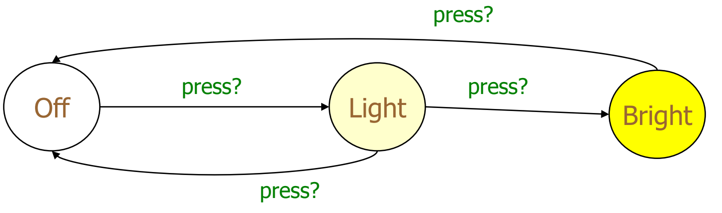
How to construct Model ?

Program as State Machine!



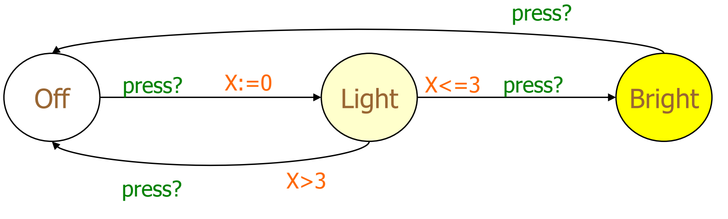
Control states

A Light Controller



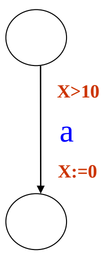
WANT: if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.

A Light Controller (with timer)



Solution: Add real-valued clock x

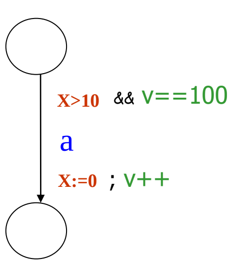
Modeling Real Time Systems



Timed Automaton

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

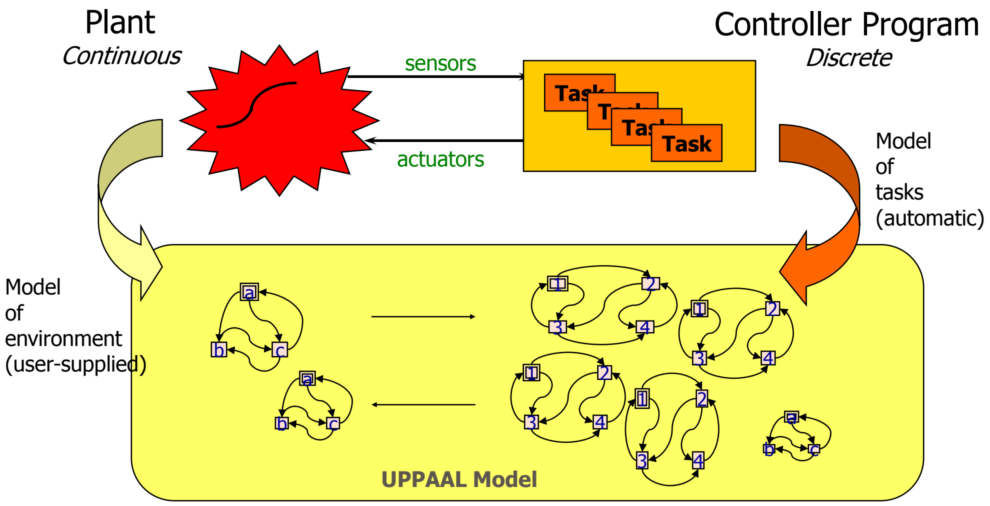
Modeling Real Time Systems

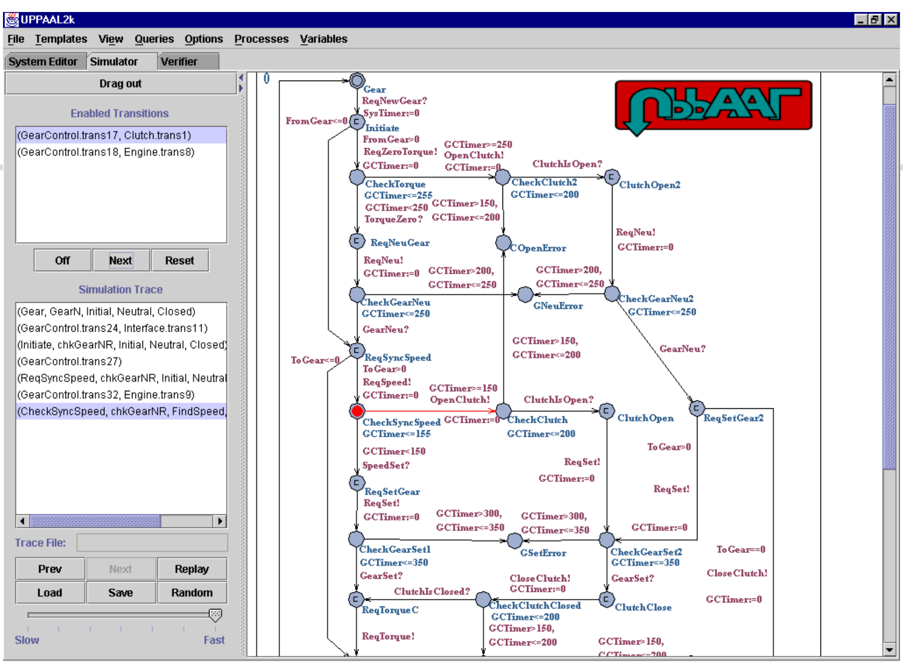


Timed Automaton in UPPAAL

- 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?

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

VERIFICATION

Model meets Specs ?

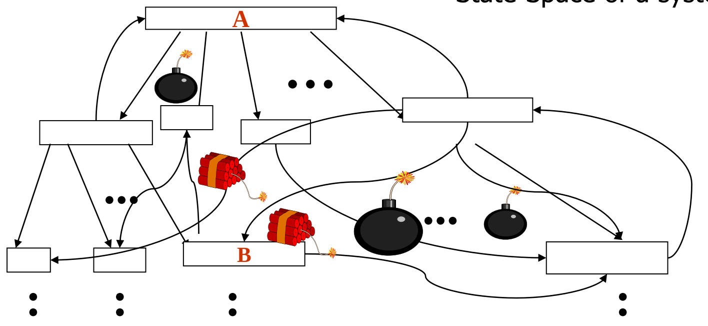
(Formal) Verification

- Semantics of a system
  = all states + state transitions
  (all possible executions)

- Verification
  = state space exploration + examination

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)?

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, Coq
  - Require too much expertise ☹
- Automatic: Model-Checking ☺
  - State-Space Exploration and Examination
  - e.g. SPIN, SMV, UPPAAL

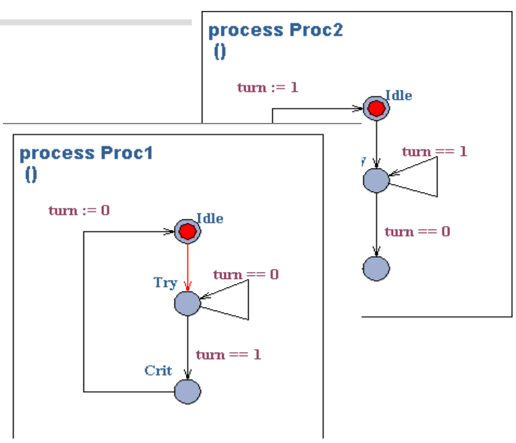
Two basic verification algorithms

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

Modelling in UPPAAL: example

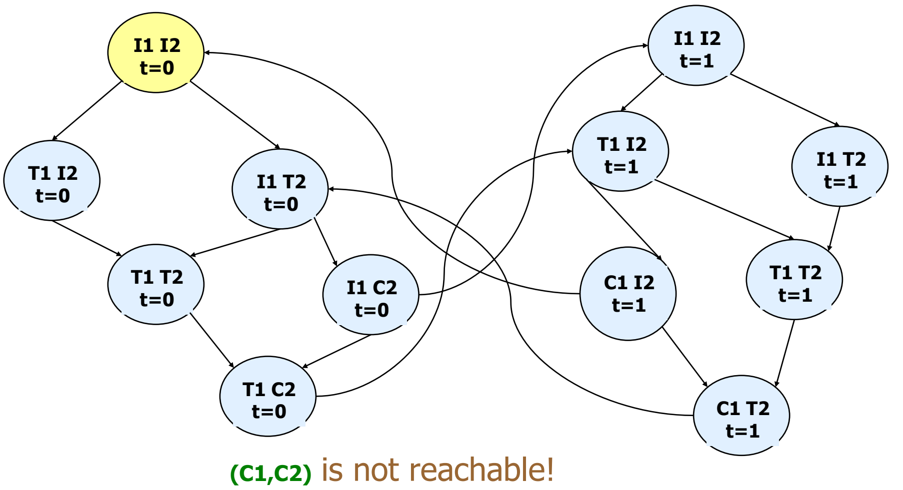
```
P1 :: while True do
      T1 : wait(turn=1)
      C1 : CS1; turn:=0
      endwhile
||
P2 :: while True do
      T2 : wait(turn=0)
      C2 : CS2; turn:=1
      endwhile
```

Mutual Exclusion Program



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

Verification: example

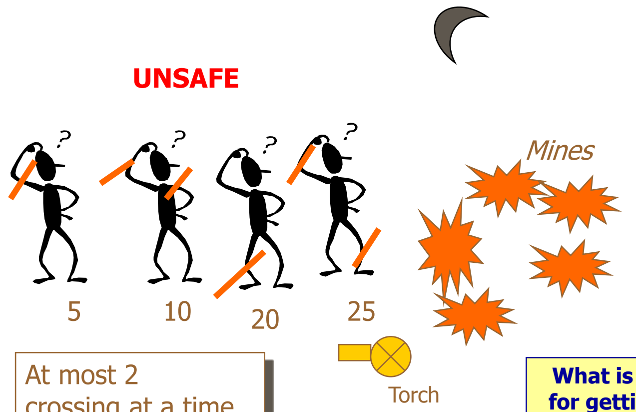


(C1,C2) is not reachable!

UPPAAL Demo

Example: the Vikings Problem

Real time scheduling

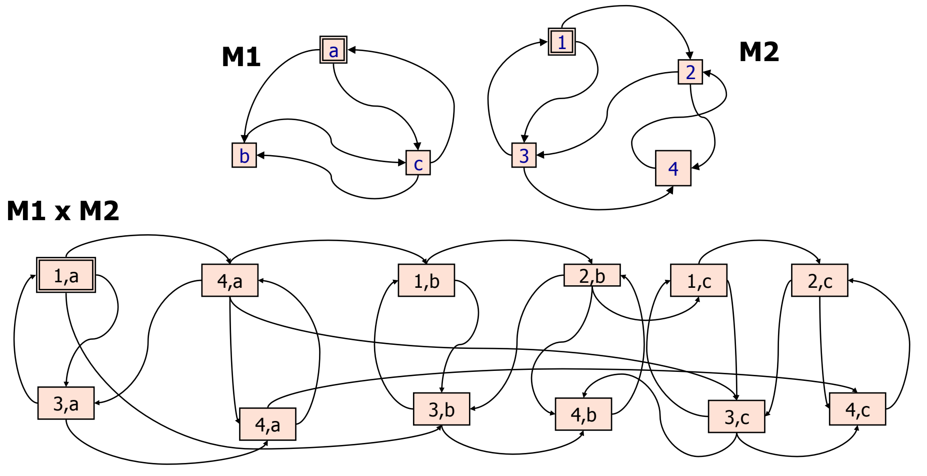


At most 2
crossing at a time
Need torch

What is the fastest time for getting all vikings to the safe side ?

This sounds too good!
What’s the problem?

Problem with verification: 'State Explosion'



All combinations = exponential in no. of components

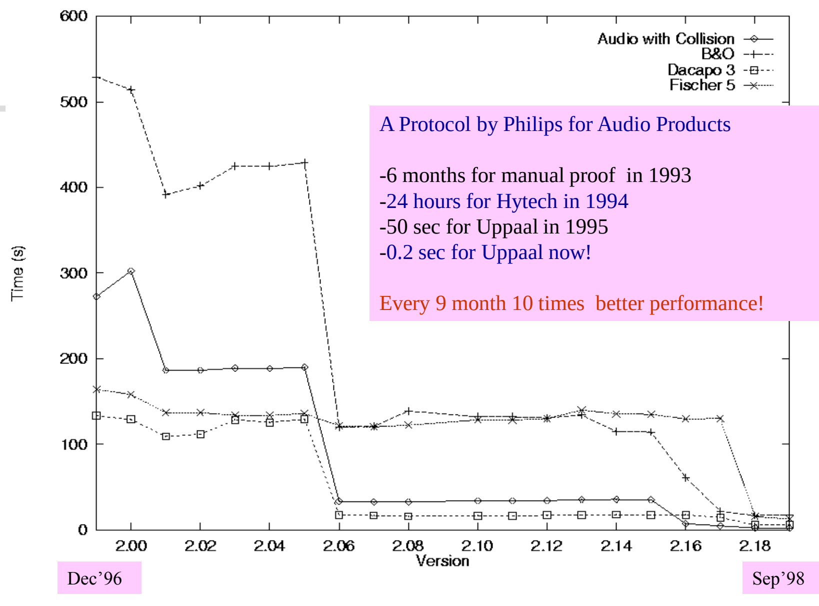
EXAMPLE

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



A Protocol by Philips for Audio Products

-6 months for manual proof in 1993
-24 hours for Hytech in 1994
-50 sec for Uppaal in 1995
-0.2 sec for Uppaal now!

Every 9 month 10 times better performance!

Dec'96 Sep'98

The dream goes on ... ...

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

End of introduction