Formal Verification, Model Checking

Radek Pelánek

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

EVROPSKÁ UNIE esf MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY OP Vzdělávání pro konkurenceschopnost UNIVERSITAS MASARYKIANA BRUNENSIS

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Formal Methods: Motivation

- examples of what can go wrong – first lecture
- non-intuitiveness of concurrency (particularly with shared resources)
  - mutual exclusion
  - adding puzzle

Formal Methods

Formal Methods

'Formal Methods' refers to mathematically rigorous techniques and tools for

- specification
- design
- verification

of software and hardware systems.

Formal Verification

Formal Verification

Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property.

Formal Verification vs Testing

	formal verification	testing
finding bugs	medium	good
proving correctness	good	-
cost	high	small

Types of Bugs

	likely	rare
harmless	testing	not important
catastrophic	testing, FV	FV

Formal Verification Techniques

manual	human tries to produce a proof of correctness
semi-automatic	theorem proving
automatic	algorithm takes a model (program) and a property; decides whether the model satisfies the property

We focus on automatic techniques.

Application Domains of FV

- generally safety-critical systems: a system whose failure can cause death, injury, or big financial loses (e.g., aircraft, nuclear station)
- particularly embedded systems
  - often safety critical
  - reasonably small and thus amenable to formal verification

Well Known Bugs

Ariane 5 explosion on its first flight; caused by reuse of some parts of a code from its predecessor without proper verification

Therac-25 radiation therapy machine; due to a software error, six people are believed to die because of overdoses

Pentium FDIV design error in a floating point division unit; Intel was forced to offer replacement of all flawed processors

Outlook

- this lecture (foundations):
  - basics of a model checking technique
  - overview of modeling formalisms, logics
  - basic algorithms
- next lectures (real-time, applications):
  - theory: timed automata
  - extensions for practical modeling
  - verification tool Uppaal
  - case studies, realistic examples

Goal of the Lecture

- goal: to understand the basic principles of model checking technique
- important for efficient use of a model checking tool

Overlap with Other Courses

- IV113 Introduction to Validation and Verification
- IA159 Formal Verification Methods
- IA040 Modal and Temporal Logics for Processes
- IA006 Selected topics on automata theory

verification in this course:

- foundations only briefly
- real-time aspects

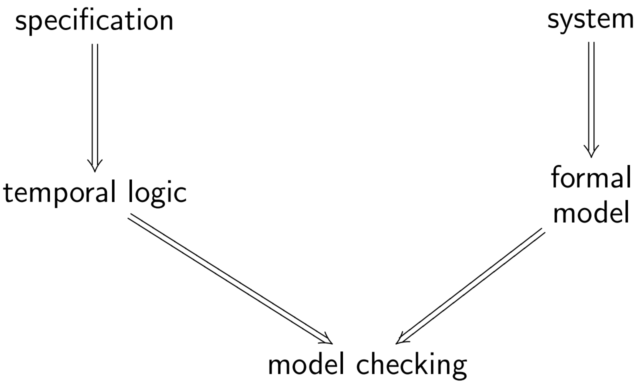
Contents

2. Modeling
   - Guarded Command Language
   - Finite State Machines
   - Other Modeling Formalisms
3. Specification
   - Types of Properties
   - Temporal Logics
   - Timed Logics
4. Algorithms
   - State Space Search
   - Logic Verification
   - State Space Explosion

Model Checking

- automatic verification technique
- user produces:
  - a model of a system
  - a logical formula which describes the desired properties
- model checking algorithm:
  - checks if the model satisfies the formula
  - if the property is not satisfied, a counterexample is produced

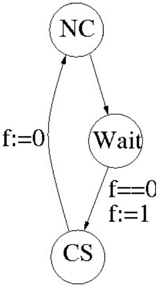
Model Checking (cont.)

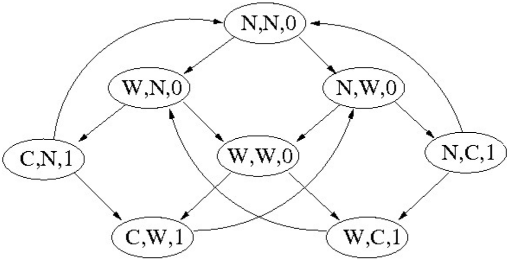


State Space

- model checking algorithms are based on state space exploration, i.e., "brute force"
- state space describes all possible behaviours of the model
- state space $\sim$ graph:
  - nodes = states of the system
  - edges = transitions of the system
- in order to construct state space, the model must be closed, i.e., we need to model environment of the system

Example: Model and State Space





Model Checking: Steps

1. modeling: system ⟶ model
2. specification: natural language specification ⟶ property in formal logic
3. verification: algorithm for checking whether a model satisfies a property

Modeling Formalisms

guarded command language	simple low level modeling language
finite state machines	usually extended with variables, communication
Petri Nets	graphical modeling language
process algebra	infinite state systems
timed automata	focus of the next lecture

Guarded Command Language

- the simplest modeling language
- not useful for actual modeling
- simple to formalize
    - we discuss formal syntax and semantics
    - foundation for later discussion of timed automata

Guarded Command Language

- integer variables
- rules:

    `if` condition `then` update
- conditions: boolean expressions over variables
- updates: sequences of assignments to variables

Example

$$a : \textit{if} \;\; x = 0 \;\; \textit{then} \;\; x := 1$$
$$b : \textit{if} \;\; y < 2 \;\; \textit{then} \;\; y := y + 1$$
$$c : \textit{if} \;\; x = 1 \wedge y \geq 1 \;\; \textit{then} \;\; x := 0, z := 1$$

Notes:

- this is an artificial example (does not model anything meaningful)
- a, b, c are names of actions
- no control flow
- rules executed repeatedly
- initial state: $x = 0, y = 0, z = 0$

Syntax

- let V be a finite set of integer variables
- expressions over V are defined using standard boolean $(=, <)$ and binary $(+, -, \cdot, ...)$ operations
- model is a tuple $M = (V, E)$
- $E = \{t_1, \ldots, t_n\}$ is a finite set of transitions, where $t_i = (g_i, u_i)$:
  - predicate g_i (a boolean expression over V)
  - update $u_i(\vec{x})$ (a sequence of assignments over V)

Semantics

The semantics of model M is a state space (formally called *Kripke structure*) $[\![M]\!] = (S, \rightarrow, s_0, L)$ where

- states S are valuations of variables, i.e., $V \rightarrow \mathbb{Z}$
- $s \rightarrow s'$ iff there exists $(g_i, u_i) \in T$ such that $s \in [\![g_i]\!], s' = u_i(s)$
  - semantics $[\![g_i]\!]$ of guards and $u_i(s)$ is the natural one
- s_0 is the zero valuation $(\forall v \in V : s_0(v) = 0)$

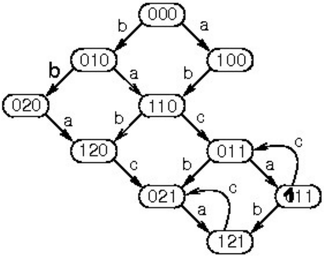
Example

$$a : \mathit{if}\ \ x = 0\ \ \mathit{then}\ x := 1$$
$$b : \mathit{if}\ \ y < 2\ \ \mathit{then}\ y := y + 1$$
$$c : \mathit{if}\ \ x = 1 \wedge y \geq 1\ \ \mathit{then}\ x := 0, z := 1$$

Construct the state space.

Example

a : *if* $x = 0$ *then* $x := 1$
b : *if* $y < 2$ *then* $y := y + 1$
c : *if* $x = 1 \wedge y \geq 1$ *then* $x := 0, z := 1$



Application

- simple to formalize, powerful (Turing power)
- not suitable for “human” use
- some simple protocols can be modeled
- control flow – variable pc (program counter)

Example: Ticket Protocol

THE SYSTEM WITH n PROCESSES

Program

global var $s, t : integer$;

begin

$t := 0$;

$s := 0$;

$P_1 \mid \ldots \mid P_n$;

end.

THE i-th COMPONENT

Process P_i ::=

local var $a : integer$;

repeat forever

think : $\langle\ a := t;$

$t := t + 1;\ \rangle$

wait : **when** $\langle\ a = s\ \rangle$ **do**

use : CRITICAL SECTION

$\langle\ s := s + 1;\ \rangle$

end.

Example: Ticket Protocol

```
pc1 := 0; pc2 := 0;
t := 0; s := 0; a1 := 0; a2 := 0;

pc1 = 0 -> pc1 := 1, a1 := t, t := t + 1;
pc1 = 1 && a1 <= s -> pc1 := 2;
pc1 = 2 -> pc1 := 0, s := s + 1;

pc2 = 0 -> pc2 := 1, a2 := t, t := t + 1;
pc2 = 1 && a2 <= s -> pc2 := 2;
pc2 = 2 -> pc2 := 0, s := s + 1;
```

Extended Finite State Machines

- each process (thread) is modelled as one finite state machine (machine state = process program counter)
- machines are extended with variables:
  - local computation: guards, updates
  - shared memory communication
- automata can communicate via channels (with value passing):
  - handshake (rendezvous, synchronous communication)
  - asynchronous communication via buffers

Example: Peterson's Algorithm

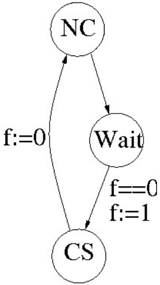
- flag[0], flag[1] (initialed to false) — meaning *I want to access CS*
- turn (initialized to 0) — used to resolve conflicts

```
Process 0:
while (true) {
   <noncritical section>;
   flag[0] := true;
   turn := 1;
   while flag[1] and
         turn = 1 do { };
   <critical section>;
   flag[0] := false;
}
```

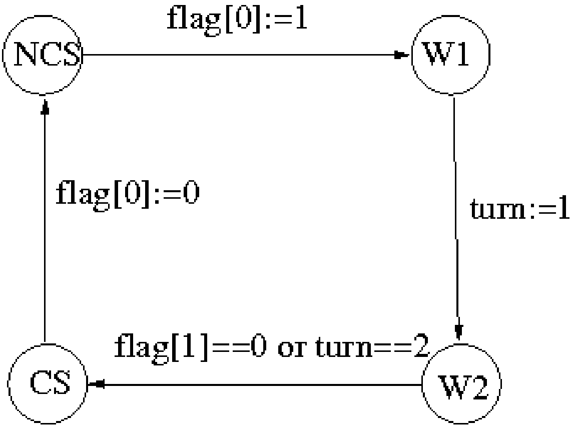
```
Process 1:
while (true) {
   <noncritical section>;
   flag[1] := true;
   turn := 0;
   while flag[0] and
         turn = 0 do { };
   <critical section>;
   flag[1] := false;
}
```

Example: Peterson's Algorithm

Exercise: create a model of Peterson's Algorithm using extended finite state machines, i.e., of the following type:



Example: Peterson's Algorithm



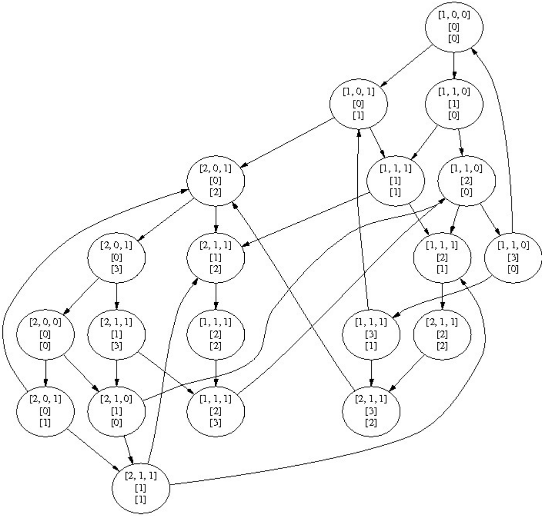
Art of Modeling

- choosing the right level of abstraction
- depends on purpose of the model, assumption about the system, ...
- example: `if x == 0 then x := x + 1`
  - one atomic transition
  - two transitions: test, update (allows interleaving)
  - multiple "assembler level" transitions: if, load, add, store

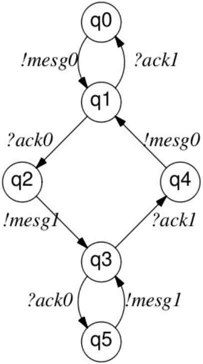
EFSM: Semantics

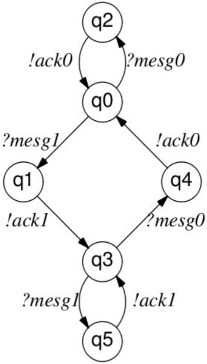
- formal syntax and semantics defined in similar way as for guarded command language
- just more technical, basic idea is the same
- note: state space can be used to reason about the model – e.g., to prove mutual exclusion requirements (cf. Assignment 1)

Example: Peterson's Algorithm



Example: Communication Protocol





Example: Elevator

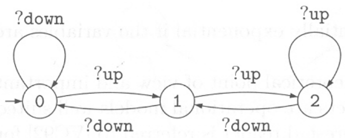


Fig. 1.13. The cabin

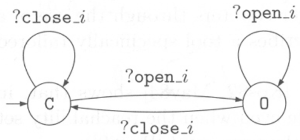
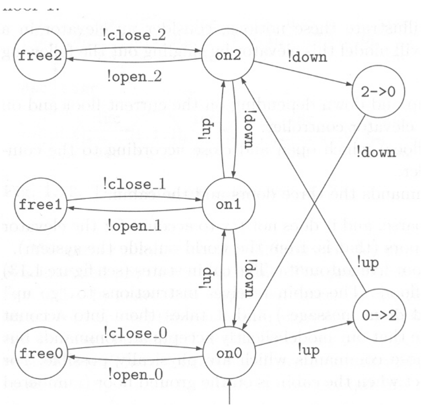
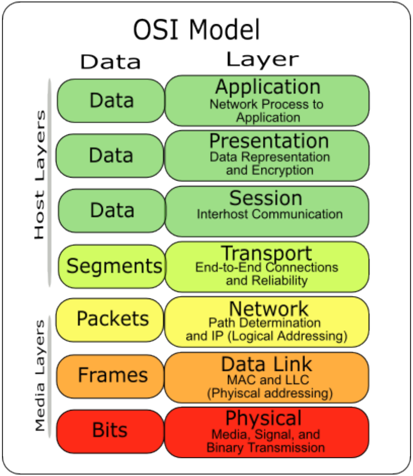


Fig. 1.14. The i^{th} door

Example: Elevator



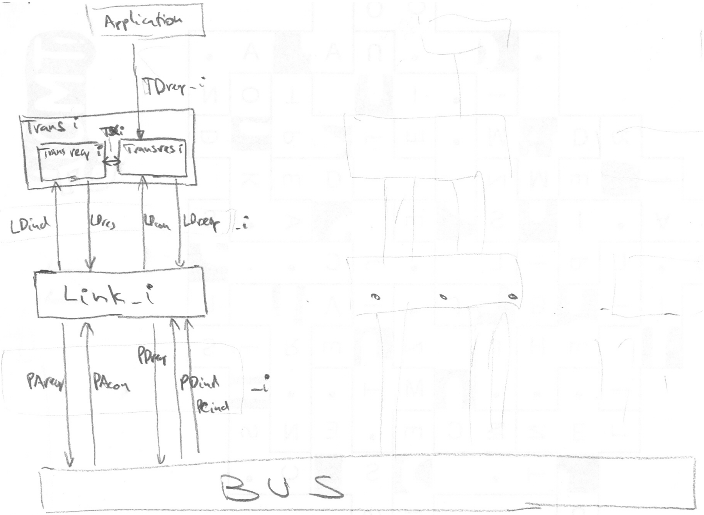
Application: Verification of Link Layer Protocol



Layer Link Protocol of the IEEE-1394

- model of the “FireWire” high performance serial bus
- *n* nodes connected by a serial line
- protocol consists of three stack layers:
  - the transaction layer
  - the link layer
  - the physical layer
- link layer protocol – transmits data packets over an unreliable medium to a specific node or to all nodes (broadcast)
- transmission can be performed synchronously or asynchronously

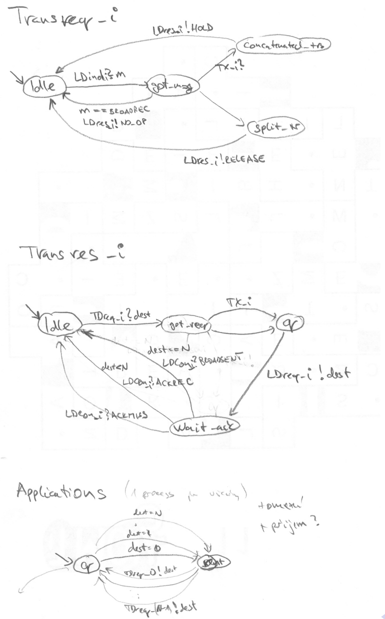
Finite State Machines



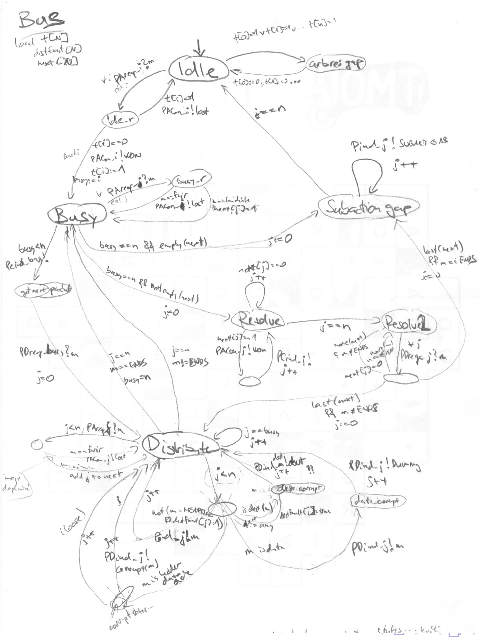
Notes

- link layer
  - main focus of verification
  - modeled in high detail
- transportation layer, physical layer (bus)
  - "environment" of link layer
  - modeled only abstractly

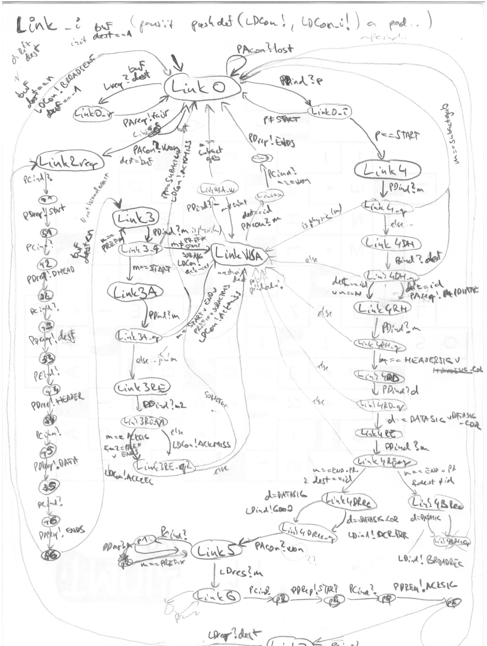
Finite State Machines



Finite State Machines



Finite State Machines

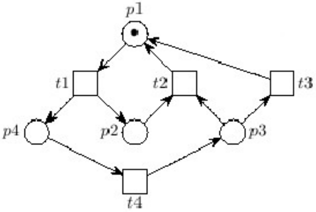


Timed Automata

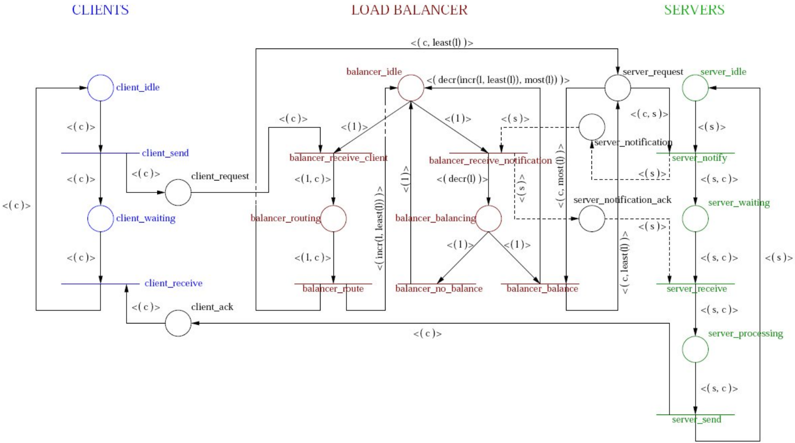
- extension of finite state machines with clocks (continuous time)
- next lecture

Petri Nets: Small Example

graphical formalism (place, transitions, tokens)



Petri Nets: Realistic Model



Process Algebra

- $A \xrightarrow{a} XX$
  $X \xrightarrow{b} A \parallel B$
- basic process algebra (BPA), basic parallel processes (BPP)
- infinite state system modeling (e.g., recursion)
- mainly theoretical research

Specification of Properties

- properties the verified system should satisfy
- expressed in a formal logic

Safety and Liveness

safety	liveness
"*nothing bad ever happens*"	"*something good eventually happens*"
example: error state is never reached	example: when a request is issued, eventually a response is generated
verification = reachability problem, find a run which violates the property	verification = cycle detection, find a run in which the 'good thing' is postponed indefinitely

Examples of Safety Properties

- no deadlock
- mutual exclusion is satisfied
- a corrupted message is never marked as a good one
- the wheels are in a ready position during the landing

Examples of Liveness Properties

- each process can eventually access critical section
- each request will be satisfied
- a message is eventually transmitted
- there will be always another sunrise

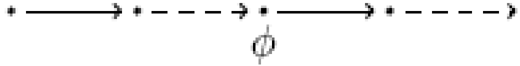
Temporal Logic

- temporal logic is a formal logic used to reason about sequences of events
- there are many temporal logics (see the course IA040)
- the main classification: linear X branching

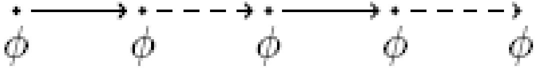
Linear Temporal Logic (LTL)

X ϕ neXt

F ϕ Future



G ϕ Globally



ψ **U** ϕ Until

LTL: Examples

a message is eventually transmitted	$\mathbf{F}\ \mathit{transmit}$
each request will be satisfied	$\mathbf{G}\ (\mathit{request} \Rightarrow \mathbf{F}\ \mathit{response})$
there will be always another sunrise	$\mathbf{G}\ \mathbf{F}\ \mathit{sunrise}$
the road will be dry until it rains	$\mathit{dry}\ \mathbf{U}\ \mathit{rains}$
process waits until it access CS	$\mathit{wait}\ \mathbf{U}\ \mathit{CS}$

LTL: Examples

What is expressed by these formulas? For each formula draw a sequence of states such that the formula is a) satisfied, b) not satisfied.

- $\mathbf{GF}a$
- $\mathbf{FG}a$
- $\mathbf{G}(a \Rightarrow \mathbf{F}b)$
- $a\mathbf{U}(b\mathbf{U}c)$
- $(a\mathbf{U}b)\mathbf{U}c$

Timed Logics

- classical temporal logics
  - good for reasoning about sequences of states
  - may be insufficient for dealing with real time
- real time extensions

Metric Interval Temporal Logic (MITL)

- extension of LTL
- temporal operator can be restricted to certain interval
- examples:
  - $\mathbf{G}(\mathit{req} \Rightarrow \mathbf{F}_{\leq 3}\mathit{serv})$
    any request will be serviced within three time units
  - $\mathit{dry}\ \mathbf{U}_{[12,14]}\ \mathit{rains}$
    after lunch it will rain, until that the road will be dry

Specification in Practice

- timed logics – mainly theoretical research
- practical specification of properties:
    - classical temporal logics
    - often limited subset or only specific patterns

State Space Search

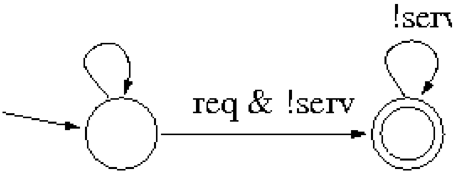
- construction of the whole state space
- verification of simple safety properties (e.g., mutual exclusion) = basically classical graph traversal (breadth-first or depth-first search)
- graph is represented implicitly = constructed on-demand from the model (description)

Logic Verification

- transformation to automata
- Buchi automaton: finite automaton over infinite words
- a word is accepted if the run of the automaton visits an accepting state infinitely often (compare with a final state for finite words)

Example

property: $\mathbf{G}(req \Rightarrow \mathbf{F}serv)$
negation: $\mathbf{F}(req \wedge \mathbf{G}\neg serv)$



Product Automaton

- property $\phi \longrightarrow$ automaton for the negation of the property $A_{\neg\phi}$
- state space of the model S + automaton $A_{\neg\phi} \longrightarrow$ product automaton $S \times A_{\neg\phi}$
- product automaton represents erroneous runs

Product Automaton: Emptiness Check

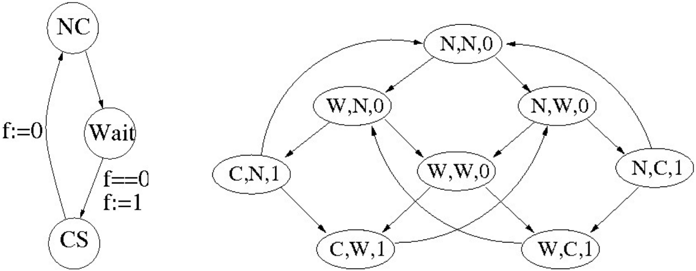
model satisfies property $\Leftrightarrow$ the language of the product automaton is empty

- verification is reduced to non-emptiness check of product automaton
- Buchi automata: non-emptiness check is performed by (accepting) cycle detection

State Space Explosion

- size of the state space grows very quickly (with respect to size of the model)
- the worst case: exponential increase (next slide)
- theory: most interesting model checking problems are PSPACE-complete
- practice: the worst case does not occur, nevertheless memory/time requirements are very high

Example



For n processes the number of states is $2^n + n \cdot 2^{n-1}$.

Dealing with State Space Explosion

- abstraction
- reduction techniques
- efficient implementations

Abstraction

- data abstraction (e.g., instead of $\mathbb{N}$ use $\{blue, red\}$)
- automated abstraction
- abstract - model check - refine

Reduction Techniques

- symmetry – consider only one of symmetric states
- partial order – consider only one of equivalent interleavings
- compositional construction – build the state space in steps

Efficient Implementations

- efficient representation of states, sets of states (symbolic methods — Binary Decision Diagrams)
- low level optimizations (e.g. memory management)
- distributed algorithms on networks of workstations
- randomization, heuristics – guiding toward errors

Model Checking: History

- 80': basic algorithms, automata theory, first simple tools, small examples
- early 90': reduction techniques, efficient versions of first tools, applications to protocol verification
- late 90': extensions (timed, probabilistic), first commercial applications for hardware verification
- state of the art: automatic abstraction, combination with other techniques, research tools for software verification, hardware verification widely adopted

Summary

- formal verification
- model checking: modeling, specification, verification
- modeling formalisms: guarded command language, finite state machines, Petri nets, ...
- formal property specification: temporal logics
- algorithms: state space search, Buchi automata, techniques for reducing state space explosion