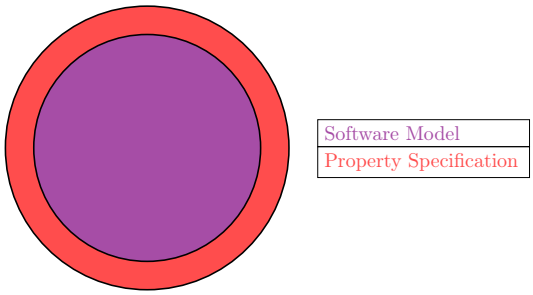
☉_☉ *Find any errors? Please send them back, I want to keep them!*

Basic Principle

given:

- Software Model M
- Property Specification S

does $M \models S$



Classification of Software Systems

Transformational Systems transforms set of (empty) input data into output data, function from state S_i to S_k
correctness:

- termination
- correctness of function $S_i \rightarrow S_k$
- correctness of input - output transformation

Reactive Systems ongoing interaction with environment, driven by environment
correctness:

- non-termination (normally)
- correctness of stimuli-response pairs

Embedded Systems usually reactive, directly connected to hardware

Cyber-Physical Systems integration of computation and physical processes

Real-Time Systems systems, where correctness depends on time a result is delivered

Soft Real-Time Systems missed deadlines will decrease result, not lead to failure, propabilities
Hard Real-Time Systems missed deadlines will lead to failure

Hybrid Systems state is characterized by discrete and continuous variables

Safety-Critical Systems systems failure may entail, death, serious injury, environmental harm, damage to property/assets

Requirement Specification

Natural Language

- ✚ very expressive
- ✚ understood by all parties

- ↯ ambiguous

Formal Language

- ✚ unambigous
- ✚ machine-analyzeable
- ↯ limited expressiveness
- ↯ hard to understand

Software Verification Method

Requirements

- formal foundation (automatic procedures)
- should be capable of relating artifacts from different stages in design cycle (formal vs informal req, design vs req, ...)
- should be easy to integrate in design cycle (high degree of automation, low degree of interaction)
- scalable

Model Checking Process

1. provide model (e.g. Promela), involves abstraction
2. simulate (check if model does, what you want)
3. elicit and formalize requirements ⇒ property specification
4. model execution: run model checker with model and specification
   outcome:
   - property is valid (check next property)
   - property is invalid (counterexample, check model for errors, check properties for errors, rethink design)
   - exhaust of memory (use more abstraction, state space reduction, incomplete methods)
   - exhaust of time (smaller model, faster computer)

Spin

- designed for communication protocols
- Open Source
- still in development
- Promela (PROtocol/PROcess MEta LAnguage)
  - concurrent modeling language
  - guarded commands
  - modeling of reactive systems

State-Based Modeling

State

- salient features of a system at given point of observation
- states can be observed, as long as features of interest unchanged

- features of interest

  point of control "program counter" (of all processes)

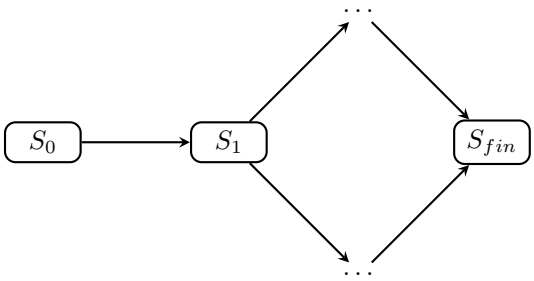
  values of local and global variables

  communication channels (messages sent but not received)

- state vector (byte-wise representation of features of interest)

State Transition

- instantaneous change of observed features of the system ("something happens")
- represents computation step
- sequence of state transitions characterize system computation



- in real-time systems, time passes in states
- in stochastic systems, transitions are labeled with probability distributions
- in hybrid systems, continuous and discrete state variables change during state transition
- transitions show the possible events
- transition sequences are valid computations sequences
- transitions encode history (S_0 has been visited before S_1)

Devising State-Machines

- "programm a model"
- abstraction: focus on relevant, can lead to non-determinism
- simplicity: find most simple abstraction that still reveals phenomena

Deadlock

- concurrent processes wait for each other in a circular wait with no pre-emption
- highly undesired

Closed System Modeling

- model checker can only validate all possible system executions under assumed environment
- model includes also environment

Transition Systems

Transition System TS is a tuple $(S, Act, \rightarrow, I, AP, L$ where

S	is a set of states
Act	is a set of actions
$\rightarrow \subseteq S \times Act \times S$	is a transition relation
$I \subseteq S$	is a set of initial states
AP	is a set of atomic propositions
$L : S \rightarrow 2^A P$	is a labeling function

- where S and Act are finite or countably infinte
- we write $s \rightarrow^{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$
- Atomic Propositions: Facts that we want to observer/that are observable in the system in any given state.
- Labeling Function: shows, which atomic propositions hold in given state.

$$Post(s, \alpha) = \left\{ s' \in S | s \xrightarrow{\alpha} s' \right\} \qquad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

$$Pre(s, \alpha) = \left\{ s' \in S | s' \xrightarrow{\alpha} s \right\} \qquad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$$

$$Post(C, \alpha) = \bigcup_{s \in C} Post(s, \alpha), \qquad Post(C) = \bigcup_{s \in C} Post(s) \text{ for } C \subseteq S$$

$$Pre(C, \alpha) = \bigcup_{s \in C} Pre(s, \alpha), \qquad Pre(C) = \bigcup_{s \in C} Pre(s) \text{ for } C \subseteq S$$

a state is **terminal** or **final** iff $Post(s) = \varnothing$

0.0.1 (Non)Determinism

A transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is **action-deterministic**, iff for all s, α

- $|I| \leq 1$ and
- $|Post(s), \alpha)| \leq 1$

else it is **action-nondeterministic**

there is at most one outgoing transition from each state labeled α

A transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is **AP-deterministic**, iff for all $s, A \in 2^{AP}$

- $|I| \leq 1$ and
- $|Post(s) \cap \{s' \in S | L(s') = A\}| \leq 1$

else it is **AP-nondeterministic**

every successor of a state s has a unique AP labeling

- Nondeterminism can lead to potentially smaller representation.
- in Software Engineering/Modeling: Abstraction
  - avoid overspecification
  - what does the system do, not how is it done
  - concurrency
    - simulate concurrency by nondeterminsm
    - either nondeterministic action can be executed first

System Execution

Given a transition system $TS = (S, Act, \rightarrow, I, AP, L)$

finite execution fragment ϱ of TS is an alternating sequence of states and actions ending with a state:

$$\varrho = s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n \text{ s.t. } s_i \rightarrow^{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i < n$$

infinite execution fragment ϱ of TS is an infinite, alternating sequence of states and actions:

$$\varrho = s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n \text{ s.t. } s_i \rightarrow^{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i$$

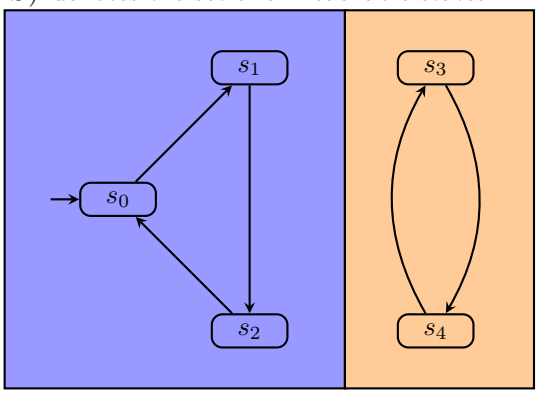
execution of TS is an inital, maximal execution fragment

- an execution fragment is maximal, iff it is either
  - finite and ending in a terminal state

  or
  - infinite
- an execution fragment is initial, iff $s_0 \in I$

reachable state is a state $s \in S$ if there exists an initial, finite execution fragment $s_0\alpha_1 s_1 \ldots \alpha_n s_n$ so that $s_n = s$

$Reach(TS)$ denotes the set of all reachable states in TS



$Reach(TS)$ not reachable

Program Graphs PG

- want to include things like variables, assignements, etc ⇒ Program Graphs
- introduce conditional transitions (transition can only be exectured, if condition is true)

$$s \xrightarrow{g:\alpha} s'$$

  - g: a boolean condition on data variables ("guard")
  - α: an action, that is possible, if g is satisfied
- assume domain of variables as infinite
  - practical implementations use variables over finite domains
- Stepwise unfolding of PGs lead to TS

valuation: assign values to variables (e.g. $\eta(x) = 17$)

propositional logic formulae

effect of actions:

$$Effect : Act \times Eval(Var) \rightarrow Eval(Var)$$

Effects define **operational semantics**

A **program graph** PG over set Var of typed variables is a tuple $(Loc, Act, Effect, \rightarrow, Loc_0, g_0)$ where

- Loc is a set of locations with initial locations $Loc_0 \subseteq Loc$
- Act is a set of actions
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function
- $\rightarrow \subseteq Loc \times (Cond(Var) \times Act) \times Loc$, transition relation cond = boolean condition
- $g_0 \in Cond(Var)$ is the inital condition

Notation $\ell \xrightarrow{g:\alpha} \ell'$ denotes $(\ell, g, \alpha, \ell') \in \longrightarrow$

Semantics for Program Graphs

- Unfolding: Transformation of PG to equivalent TS
  - states:
    - state $< l, \eta \rangle$: current control location l + data valuation η
    - initial state: initial location satisfying condition g_0
  - propositions:
    - at_l: control is at location l
    - $x \in D$ iff $D \subseteq dom(x)$
  - labeling
    - $< l, \eta \rangle$ is labeled with at_l and all conditions that hold in η
- from transitions in PG to transitions in TS
  - if $\ell \xrightarrow{g:\alpha} \ell'$ and g holds in η, then $< l, \eta \rangle \rightarrow^{\alpha} < l', Effect(\alpha, \eta) \rangle$

Structured Operational Semantics Sos

- definition of **operational semantics** of a program in terms f computations steps defined by a transition system
- whether a step happens is determined by inference rules:

$$\frac{premise}{conclusion}$$

  - if the premise holds, the conclusion holds (and can trigger further inference rules)
  - if the premise is a **tautology**, it may be omited
    the rule is then called an **axiom**
- semantics is structural, because it applies inference rules recursively to syntactic structure

Transition Systems for Program Graphs

The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \longrightarrow, Loc_0, g_0)$$

over set Var of variables is the tuple

$$(S, Act, \longrightarrow, I, AP, L)$$

where

- $S = Loc \times Eval(Var)$
- $\longrightarrow \subseteq S \times Act \times S$ is defined by $\dfrac{\ell \xrightarrow{g:\alpha} \ell' \ \wedge \ \eta \models g}{< \ell, \eta > \xrightarrow{\alpha} < \ell', Effect(\alpha, \eta) >}$
- $I = \{< \ell, \eta > | \ell \in Loc_0, \eta \models g_0\}$
- $AP = Loc \cup Cond(Var)$ and $L(< \ell, \eta >) = \{\ell\} \cup \{g \in Cond(Var) | \eta \models g\}$

Data in Transition Systems

- TS do not possess data variables, data values and their changes need to be encoded in states
- assume $i = 1, \ldots, n$ variables of domain $s_i \Rightarrow \Pi_{i=1}^{n} s_i$
  combinatorial, exponential state space explosion
- variables over infinite domains ⇒infinite number of states

Modelling Concurrency

Concurrent Event

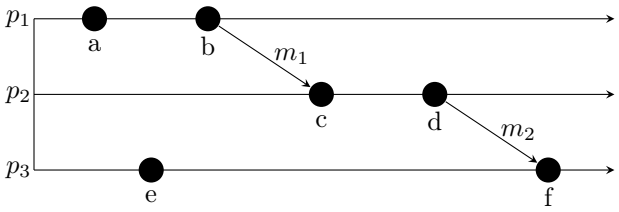
- logically independent events in separate autonomous system

Lamport's happened-Before Relation $\rightarrow$

HB1: for any pair of events e and e', if there is a process p_i such that $e \rightarrow_i e'$, then $e \rightarrow e'$

HB2: for any pair of events e and e' and for any message m, if $e = send(m)$ and $e' = receive(m)$, then $e \rightarrow e'$

HB3: if e, e' and e'' are events and if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$ (HB is identical to its transitive closure)



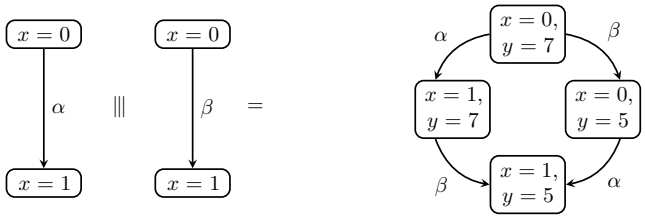
Concurrency

- two events are E_1, E_2 are concurrent, if each ordering of them is possible

Interleaving

- concurrent actions α, β are executed in arbitrary order
- $Effect(\alpha \ ||| \ \beta, \nu) = Effect((\alpha; \beta) + (\beta; \alpha), \nu$
  - $|||$ is binary interleaving operator
  - ; is sequential execution
  - $+$ is nondeterministic choice

– $\underbrace{x := x + 1}_{=\alpha} \;|||\; \underbrace{y := y - 2}_{=\alpha}$



Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i), i = 1, 2$ be two transition systems.

$$TS_1 \;|||\; TS_2 = (S_1 \times S_2, Act_1 \uplus Act_2), \rightarrow, I_1 \times I_2, AP_1 \uplus AP_2, L)$$

where $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$ and the transition relation $\rightarrow$ is defined by the rules:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle} \text{ and } \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1, s_2' \rangle}$$

– Interleaving of Program Graphs
let PG_1 and PG_2 be program graphs

without shared variables:
i.e. $Var_1 \cap Var_2 = \varnothing$
concurrent behaviour: $TS(PG_1) \;|||\; TS(PG_2)$

with shared variables:
concurrent behaviour: $TS(PG_1 \;|||\; PG_2)$

in general:
$TS(PG_1) \;|||\; TS(PG_2) \neq TS(PG_1 \;|||\; PG_2)$

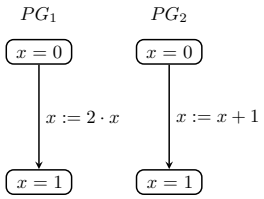
For program graph $PG_i = (Loc_i, Act_i, Effect_i, \longrightarrow_i, Loc_{0,i}, g_{0,i})$ over variables Var_i, the Program graph $PG_1 \;|||\; PG_2$ over $Var_1 \cup Var_2$ is defined by:

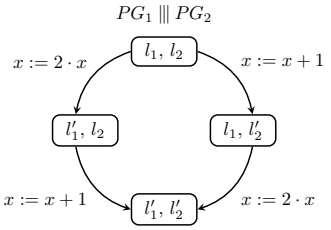
$$(Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \longrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

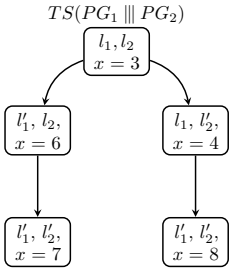
where $\longrightarrow$ is defined by the inference rules:

$$\frac{l_1 \xrightarrow{g:\alpha} l_1'}{\langle l_1, l_2 \rangle \xrightarrow{g:\alpha} \langle l_1', l_2 \rangle} \text{ and } \frac{l_2 \xrightarrow{g:\alpha} l_2'}{\langle l_1, l_2 \rangle \xrightarrow{g:\alpha} \langle l_1, l_2' \rangle}$$

and $Effect(\alpha, \eta) = Effect_i(\alpha, \eta)$ if $\alpha \in Act_i$.







– **atomicity** a sequence of statements is called **atomic**, if it cannot be interleaved with program statements from a concurrently executing program.

$$\underbrace{\langle x := x + 1; y := 2x + 1\rangle}_{atomic}; x := 0$$

Inter Process Synchronisation

- concurrent processes need to synchronize their computations
  - adjust relative speed
  - exchange data, that may be needed by other processes
- common synchronization methods:
  - shared variables
  - message passing
    - ∗ synchronous (rendez-vous, handshaking)
      wait till other party is ready

Let $TS_i = (S_i, Act_i, \longrightarrow_i, I_i, AP_i, L_i), i = 1, 2$ and $H \subseteq Act_1 \cap Act_2$.
Introducing: The **handshake-operator** $\|_H$ for an action set H in the following way:

$$TS_1 \|_H TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \longrightarrow, I_1 \times I_2, AP_1 \uplus AP_2, L)$$

where $(\langle s_1, s_2\rangle) = L_1(s_1) \cup L_2(s_2)$ and $\longrightarrow$ is defined as follows

· for $\alpha \notin H$ (interleaving)

$$\frac{s_1 \xrightarrow{\alpha} s_1'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2\rangle} \qquad \frac{s_2 \xrightarrow{\alpha} s_2'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1, s_2'\rangle}$$

· for $\alpha \in H$ (handshaking)

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \wedge s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2'\rangle}$$

· $\|$ is shorthand for $\|_H$ when $H = Act_1 \cap Act_2$

- ∗ asynchronous (typically fifo, queue)
  · $c!e$: send the value of expression e along the channel c
  · $c?x$: receive a message from channel c and assign it to variable x
  · channels have storage capacity $cap(c)$
  if $cap(c) \in \mathbb{N}$, then c is a channel with finite capacity
  if $cap(c) = \infty$, then c has infinite capacity
  $cap(c) = 0$ sometimes means, that the channel uses synchronous handshake communication
  · types of communication:
  non-blocking: execution never delays invoker
  blocking: otherwise
  · when is blocking taking place?
  asynchronous message passing with unbounded buffers ⇒sender never blocks
  synchronous message passing: no buffering
  buffered message passing with bounded, finite capacity
  sender blocks on full buffer
  sender does not block on full buffer (entails message loss)

Channel System

A **channel system** CS over $(Var, Chan)$ consists of i program graphs PG_i over $(Var_i, Chan_i)$, for $1 \leq i \leq n$, with $Var = \bigcup_{1 \leq i \leq n} Var_i$ and is denoted by $CS = [PG_1| \dots |PG_n]$
Global states are tuples of the form $\langle l_1, \dots, l_n, \eta, \xi \rangle$ where

l_i current control location of process i

η current valuation of variables

ξ current valuation of channels

- A **channel evaluation** ξ is a mapping from channel $c \in Chan$ onto sequence $\xi(c) \in dom(c)^*$ such that
  - current length cannot exceed the capacity of $c : len(\xi(c)) \leq cap(c)$
  - $\xi(x) = v_1 v_2 \dots v_k, cap(c) \geq k$ denotes, v_1 is at front of buffer, etc.
  - $\xi[c := v_1 \dots v_k]$ denotes channel evaluation

$$\xi[c := v_1 \dots v_k](c') = \begin{cases} \xi(c') & if c \neq c' \\ v_1 \dots v_k & c = c \end{cases}$$

  - initial channel evaluation ξ_0 equals $\xi_0(c) = \varepsilon$ for any c.

  Let $CS = [PG_1| \dots |PG_n]$ be a **channel system** over $(Chan, Var)$ with

$$PG_i = (Loc_i, Act_i, Effect_i,) \rightsquigarrow_i, Loc_{0,i}, g_{0,i}, \text{ for } 0 < i \leq n$$

  $TS(CS)$ is the **transition system** $(S, Act, \rightarrow, I, AP, L)$ where

  - $S = (Loc_1 \times \dots \times Loc_n) \times Eval(Var) \times Eval(Chan)$
  - $Act = (\biguplus_{0<i\leq n} Act_i) \uplus \{\tau\}$
  - $I = \{\langle \ell_1, \dots \ell_n, \eta, \xi_0 \rangle | \forall i.(l_i \in Loc_{0,i} \wedge \eta \models g_{0,i}) \wedge \forall c.\xi_0(c) = \varepsilon\}$
  - $AP = \biguplus_{0<i\leq n} Loc_i \uplus Cond(Var) >$
  - $L(\langle \ell_1, \dots, \ell_n, \eta, \xi \rangle) = \{\ell_1, \dots, \ell\} \cup \{g \in Cond(Var) | \eta \models g\}$
  - $\rightarrow$ is defined by the inference rules:

    Interleaving for $\alpha \in Act_i$:

$$\frac{\ell_i \xrightarrow{g:\alpha} \ell_i' \wedge \eta \models g}{\langle \ell_1, \dots, \ell_i, \dots, \ell_n, \eta, \xi \rangle \xrightarrow{\alpha} \langle \ell_1, \dots, \ell_i', \dots, \ell_n, \eta', \xi \rangle}, where\ \eta' = Effect(\alpha, \eta)$$

    Synchronous message passing over $c \in Chan,\ cap(c) = 0$:

$$\frac{\ell_i \xrightarrow{g:c?x} \ell_i' \wedge \ell_j \xrightarrow{g:c!e} \ell_j' \wedge \eta \models g \wedge g' \wedge i \neq j}{\langle \ell_1, \dots, \ell_i, \dots, \ell_j \dots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \dots, \ell_i', \dots, \ell_j' \dots, \ell_n, \eta', \xi \rangle}, where\ \eta' = \eta[x := \eta(e)]$$

    Asynchronous message passing over $c \in Chan, cap(c) < 0$:

$$\frac{\ell_i \xrightarrow{g:c?x} \ell_i' \wedge \eta \models g \wedge len(\xi(c)) = k > 0 \wedge \eta(c) = v_1 \dots v_k}{\langle \ell_1, \dots, \ell_i, \dots, \ell_n, \eta, \xi \rangle \xrightarrow{\alpha} \langle \ell_1, \dots, \ell_i', \dots, \ell_n, \eta', \xi' \rangle}, where\ \eta' = \eta[x := v_1] \text{ and } \xi' = \xi[c := v_2 \dots v_k]$$

    transmit value $\eta(e) \in dom(c)$ over channel c:

$$\frac{\ell_i \xrightarrow{g:c!e} \ell_i' \wedge \eta \models g \wedge len(\eta(c)) = k < cap(c) \wedge \eta(c) = v_1 \dots v_k}{\langle \ell_1, \dots, \ell_i, \dots, \ell_n, \eta, \xi \rangle \xrightarrow{\alpha} \langle \ell_1, \dots, \ell_i', \dots, \ell_n, \eta', \xi' \rangle}, where\ \xi' = \xi[c := v_1 v_2 \dots v_k \eta(e)]$$

nanoPromela

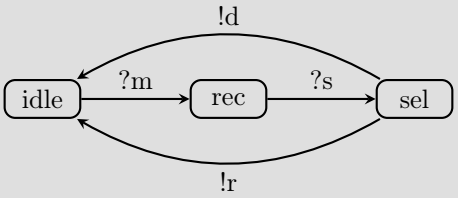
⇒5-38 to 5-50

State Machine Based Modeling

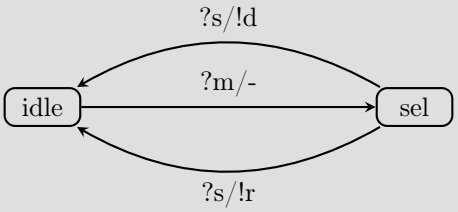
- Practical use: no TS or PG ⇒state machine models
  can all be modelled as Program Graphs, Channel Systems or Transition Systems
  UML-RT ⊂ UML : state charts
- Mealy and Moore Machines
  - expressively equivalent
  - non-deterministic

	Q :	finite set of states
	$q_0 \in Q$:	initial state
	I:	an alphabet (input symbols)
let	O with $I \cap O = \varnothing$:	an alphabet (output symbols)
	$A = I \cup O$	event alphabet
	$\delta : Q \times A \to Q$:	a relation
	$\rho : Q \times I \to O \times Q$:	a relation

Moore machine: we call (Q, q_0, A, δ) a **finite Moore machine**



Mealey machine: we call (Q, q_0, I, O, ρ) a **finite Mealey machine**

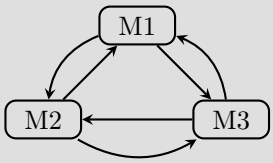


- Desiderata for Reactive System Models:
  1. representation of **data**
  2. representation of **concurrency**
  3. representation of **communication**
     especially: asynchronous communication / message passing
- Communicating Finite State Machines CFSMs
  - groups of independent, concurrently executing state machines
  - communication and synchronization by message passing over unbounded buffers
- Extended Finite State Machines EFSM
  - data variables
  - predicates
  - operations

- CFSMs after Brand and Zafiropulo

  - concurrenct FSM (≥ 2) + communication channels
  - every FSM represents a concurrent communicating process with a finite number of control states
  - every communication channel is:
    1. full duplex
    2. error-free
    3. has a first-in-first-out strategy
    4. has unbounded capacity

    1.-3. characterize a **perfect full-duplex** channel
  - one pair of channels (c_{ij} and c_{ji}) for each pair (i, j) of machines



- formalisation:

N :	a positive integer
$i, j = 1, \ldots N$:	index over processes
$\langle Q_i \rangle_{i=1}^{N}$:	N disjoint, finite sets, Q_i denots the state set of process I
$\langle A_{ij}^{N} \rangle_{i,j=1}$:	N^2 disjoint sets, with $(\forall i)(A_{ii} = \varnothing)$ A_{ij} denotes the message alphabet for the channel $i \to j$
δ:	relation, determining, for each pair i, j the following function:
	– $Q_i \times A_{ij} \to Q_i$
	– $Q_i \times A_{ji} \to Q_i$
$\langle q_i^0 \rangle$:	tupel of initial states, $(\forall i)(q_i^0 \in Q_i)$

- we call $\big(\langle Q_i \rangle, \langle q_i^0 \rangle, \langle A_{ij} \rangle, \delta\big)$ a **protocol**

$s_i \in Q_i$:	state of process i
$x_{ij} \in A_{ij}$:	a message
	– $?x_{ij}$ reception of a message
	– $!y_{ji}$ sending of a message

- Notation:

$f((s_i, \ldots, s_n)) = (f(s_i), \ldots, f(s_n))$

x, y:	message
X, Y:	sequence of messages
x, xy, xY, xXY:	concatenated sequences of messages

Formal Semantics for a Promela-like

Semantics of a protocol: set of admissable state sequences

State of a protocol?

- sum of
  - local state of each of the $1 \ldots N$ processes
  - state of all channels $c_{ij} \in A_{ij}^*$
    each c_{ij} corresponds to a sequence of messages that have been sent, but not yet received
- we call this the **global system state**

- obtain set of all computations of a protocol:
  - initially: all processes in q_i^0 and all $c_i j = \varnothing$

- system is in current state s
- state transition triggered by **send** and **receive** events

  send event: * add a message to the tail of the corresponding message queue
  * change local system state of sending process

  receive event: * take the message to be received from the head of the message queue
  * change local system state of receiving process
- leads into new global system state s'

- Global System State
  - $P = \left(\langle Q_i \rangle, \langle q_o^0 \rangle, \langle A_i j \rangle, \delta\right)$ a protocol
  - $S = (S_1, \ldots, S_N)$ an N-tuple of local process states
  - C an N^2 tuple

$$C = \begin{pmatrix} & c_1 & \cdots & c_N \\ c_1 & \epsilon & & \\ \vdots & & \ddots & \\ c_N & & & \epsilon \end{pmatrix}$$

    so that for all $i, j : c_{ij} \in A_{ij}^*$
  - we call (S, C) a **global system state**
  - **State Transition Relation** let P a protocol and $G = \{(S, C) | (S, C) \text{ is a global system state}\}$
    $\vdash$: $G \to G$ is defined as follows
    $(S, C) \vdash (S', C')$ iff $\exists\, i, k, x_{ik}$ such that either
    1. (S, C) and (S', C') are identical except for the following exceptions

$$s_i' = \delta(s_i, !x_{ik}) \text{ (sending by i)}$$

$$c_{ik}' = c_{ik} x_{ik}$$

    2. (S, C) and (S', C') are identical except for the following exceptions

$$s_k' = \delta(s_k, ?x_{ik}) \text{ (receiving by k)}$$

$$c_{ik} = x_{ik} c_{ik}'$$

  - Reachable Global System State, Paths and Acceptance
    * G^0 the initial global system state of a protocol
    * G a global system state of the same protocol
    * $\vdash$ the state transition relation of this same protocol
    * $\vdash^*$ denotes the transitive closure of $\vdash$
  - we say that G is rechable if

$$G^0 \vdash^* G$$

- When is a Problem P decidable?
  There exists an algorithm which terminates after a finite number of steps whether $P = \varnothing$ or not.
- What is a Turing machine? you should know ...
- Example of an undecidable problem?
  There is no TM that will decide whether an arbitrary given TM will halt or not.
- When is a formalism Turing-complete?
  When you can simulate a TM in this formalism.
- CFSM are Turing-complete
  - three processes: P_1, P_2, P_3
  - simulate the control of the TM in the state machine of P_2

- use P_1 and the channels c_{21} and c_{c12} to simulate the left half tape, use P_3 and c_{23} and c_{32} to simulate the right half tape
- note: all c_{ik} have unbounded length
  ⇒infinite state space (globally)
  ⇒undecidable problems:
  * termination
  * will some communication event ever be executed?
  * is some system state reachable
  * is the protocol deadlock-free?
- a channel c_{ij} is **bounded** if, for every reachable global system state (C, S), the length of c_ik is bounded by a constant h
  with bounded channels above problems are decidable, however deadlocks may be introduced due to bounded chans.

Extended Finite State Machines (EFSM)

An EFSM is a FSM extended by

- data abstraction
- operations on variables
- symbolic (explicit states)
- boolean transition conditions

A (symbolic) state of an EFSM represents an **equivalence class** of system states.

we call $E = (S, D, V, O, I, T, C)$ an extended finite state machine EFSM, where
S: set of symbolic states
D: n-dimensional linear space, each D_i is an (infinite) data domain
V $= \{\Pi, v_1, \ldots, v_n\}$: finite set of programme variables

- Π: control variable over domain S
- $\{v_1, \ldots, v_2\} \in D$: data variables

O: finite set of output signal types
I: finite set of input signal types
T: $S \times 2^D \times I \to S \times 2^D \times O$
C: an initial condition over $S \times 2^D$

Communicating Extended Finite State Machines

- foundation for many practical specifications and modeling languages
  - Specification and Description Language SDL (now part of UML)
  - Estelle (ISO)
  - ROOM/UML-RT
  - UML-RT
  - SysML

State Explosion Problem

- Size of Transition System
  - size of $TS = |S| + | \to |$
    * dependend on
      · use of data (variables) (exponentially larger)
      · use of concurrent composition

· use of communication channels

– Program Graphs
  * number of states of $TS(PG) = |\#programm\ locations| \cdot \prod_{variable\ x} |dom(x)|$
  * growth of $TS(PG)$
    · assume N variables, each with $k = |dom(x)|$ possible values
    $\Rightarrow k^N$ states exponential growth
    · Programm with 10 locations, 3 boolean variables, 5 integer variables ranging $1 - 10$
    $10 \times 2^3 \times 10^5 = 8\,000\,000 = 8 \times 10^6$ states

- Concurrent Composition of Transition Systems
  – derived form concurrent programs
  – size:
    * Cartesian product of state spaces of componens TSs, i.e. $\#states\ of\ P_1 \times \cdots \times \#states\ of\ P_2$
    * assume N components of size k each $\Rightarrow$size of $TS = k^N$

- Channel Systems:
  – assynchronous communication channels c with capacity $cap(c)$
  – assume K components and N channels
  $\Rightarrow$worst case size: $\prod_{i=1}^{N} \left(|\#programm\ locations| \prod_{variable\ x} |dom(x)| \right) \cdot \prod_{j=1}^{K} |dom(c_j)^{cap(c_j)}$

Linear-Time Properties and Invariants

Property: A system execution (computation) will be modeled as a **sequence of states or events**.

- $\sigma_0 =< g, a, z, g, \cdots >$
- $\sigma_1 =< g, a, d, g, \cdots >$

A (model of the system/system/program) P has the property Π if all its computations are in Π.

Property Representation: too cumbersome to enumerate all infinite computations $\Rightarrow$use mathematical formalisms:

- ω-automata (property corresponds to accepted language)
- ω-regular expressions
- temporal logic

State Graph

- The **state graph** of TS ($G(TS)$, is the digraph (V, E) with vertices $V = S$ and edges $E = \{(s, s') \in S \times S | s' \in Post(s)\}$ $\Rightarrow$omit all state and transition labels in TS and ignore being inital
- $Post^*(s)$ is the set of reachable states $G(TS)$ from s:

$$Post^*(C) = \bigcup_{s \in C} Post^*(s) \text{ for } C \subseteq S$$

- $Pre^*(C)$ has analogous meaning
- set of reachable states: $Reach(TS) = Post^*(I)$

Path Fragment

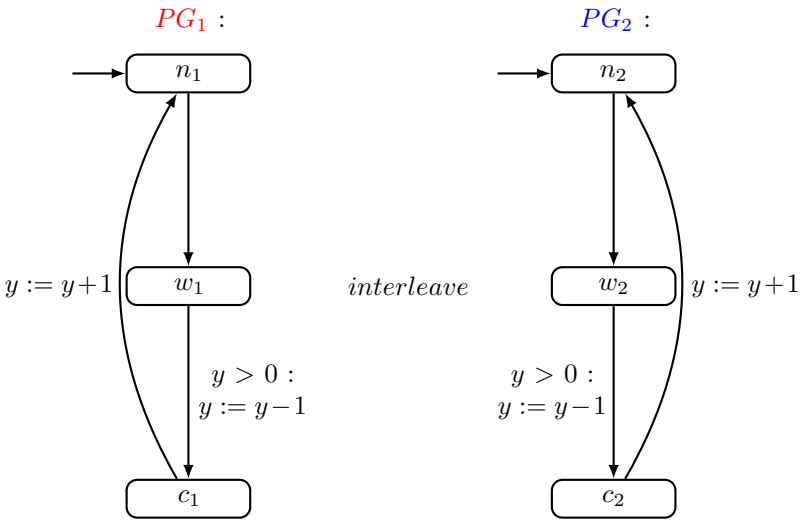
- A **path fragment** is an execution fragment without actions
- A **finite path fragment** $\hat{\pi}$ of TS is a state sequence:

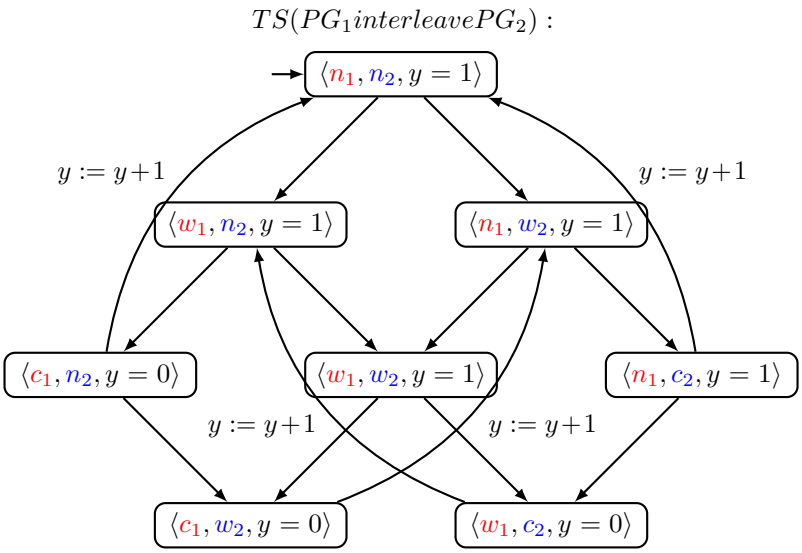
$$\hat{\pi} = s_0 s_1 \dots s_n \text{ such that } s_{i+1} \in Post(s_i) \text{ for all } 0 \leq i < n \text{ where } n \geq 0$$

- An **infinite path fragment** π of TS is an infinite state sequence:

$$\pi = s_0 s_1 \dots \text{ such that } s_{i+1} \in Post(s_i) \text{ for all } i \geq 0$$

- A **path** of TS is an initial, maximal path fragment
  - a **maximal** path fragment is either finite ending in a terminal state or infinite
  - a path fragment ist **initial** if $s_0 \in I$
  - $Paths(s)$ is the set of maximal path fragments π with $first(\pi) = s$





Traces

- Let $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states all maximal paths (and executions) are infinite
- The **trace** of the infinite path fragment $\pi = s_0 s_1 \dots$ is $trace(\pi) = L(s_0)L(s_1)\dots$
- The **trace** of the finite path fragment $\pi = s_0 s_1 \dots s_n$ is (s_n)
- The set of traces of a set Π of pahts: $trace(\Pi) = \{trace(\pi) | \pi \in \Pi\}$
-

$$Traces(s) = trace(Paths(s)) \qquad Traces(TS) = \bigcup_{s \in I} Traces(s)$$

$$Traces_{fin}(s) = trace(Paths_{fin}(s)) \qquad Traces_{fin}(TS) = \bigcup_{s \in I} Traces_{fin}(s)$$

Example $AP = \{crit_1, crit_2\}$

path: $\pi = \langle n_1, n_2, y = 1\rangle \rightarrow \langle w_1, n_2, y = 1\rangle \rightarrow \langle c_1, n_2, y = 0\rangle \rightarrow$
$\langle n_1, n_2, y = 1\rangle \rightarrow \langle n_1, 2_2, y = 1\rangle \rightarrow \langle n_1, c_2, y = 0\rangle \rightarrow \dots$

trace: $trace(\pi) = \varnothing\varnothing\{crit_1\}\varnothing\varnothing\{crit_2\}\varnothing\varnothing\{crit_1\}\varnothing\varnothing\{crit_2\}\dots$

Linear Time Properties for Transition Systems

- for a given TS a linear time property Π over AP is a subset of the set of all infinite strings formed over subset of AP: $\Pi \subseteq (2^{AP})^{\omega}$
- Π specifies the set of all admissible observable behaviour of the system
- satisfaction: $TS \models S$ iff $Traces(TS) \subseteq \Pi$
- notation: state $s \in S$ satisfies Π, written as $s \models \Pi$, iff $Traces(s) \subseteq \Pi$

Trace Equivalence

- Let TS and TS' be two transition systems over AP without terminal state

  trace inclusion $Traces(TS \subseteq Traces(TS')$ iff for any LT property P : $TS \models P$ implies $TS' \models P$
  trace equivalence $Traces(TS = Traces(TS')$ iff TS and TS' satisfy the same LT properties
- i.e. there is **no LT property** that can distinguish between TS and $TS' \Rightarrow TS$ and TS' are trace equivalent

Invariant LT properties

- Properties over AP that hold for all rechable states, including the initial state (e.g. mutual exclusion, deadlock freedom)

An LT property P_{inv} over AP is called an **invariant** if there is a propositional formula ψ over AP such that

$$P_{inv} = \{A_0, A_1, \dots \in (2^{AP})^{\omega} | (\forall j \geq 0)(A_j \models \psi)\}$$

ψ is then referred to as the **invariant condition** for invariant P_{inv}

- $TS \models P_{inv}$
  iff $trace(\pi) \in P_{inv} \forall\ \pi\ in\ TS$
  iff $L(s) \models \psi \forall$ states s that belong to a path of TS
  iff $L(s) \models \psi \forall$ states $s \in Reach(TS)$

- Model Checking Invariants
  - perform systematic forward search
    depth first search Dfs
    breadth first search Bfs
  - alternative: backwards search
    * compute all states violating invariant condition ψ
    * starting here, compute $Pre = \bigcup_{S \in \mathcal{S}, S \not\models \psi} Pre^*(s)$
    * Check if Pre contains an initial state
  - Time complexity of Dfs: $\mathcal{O}(N \times (|\Phi| + 1) + M)$
    * N: number of reachable states
    * $(|\Phi| + 1)$ length of formula
    * $M = \sum_{s \in \mathcal{S}} |Post(s)$: number of transitions in the reachable part of TS

    $\Rightarrow$time complexity of Dfs is linear
- Computation and representation of successor states:

  explicit state model checking – successor states are implicitly given (program graph, Promela code, ...)
  – $Post(s)$ is typically stored as adjacency list
  – successor states are explicitly computet (state vectors)

  symbolic model checking – $Post(s)$ is represented symbolically (binary decision diagrams (BDD))
  – reachability computation is fixed point computation

Safety and Liveness

According to Leslie Lamport

safety: something bad will never happen

liveness: something good will eventually happen

- Property P corresponds to a language
  - $P \subseteq (2^{AP})^*$: the property is finitary
  - $P \subseteq (2^{AP})^\omega$: the property is infinitary
- often convenient to express property by ω-regular expressions

Safety Properties

- An LT property P_{safe} is a **safety property**, iff $\forall \sigma \in (2^{AP})^\omega \setminus P_{safe}$ there exists a finite prefix $\hat{\sigma}$ of σ such that $P_{safe} \cap \{\sigma' \in (2^{AP})^\omega | \hat{\sigma}$ is a prefix of $\sigma'\} = \varnothing$

  If we violate the safety property, there is no way to fix it. OR: Violation can be confirmed by a finite prefix.
- $\hat{\sigma}$ is called a **bad prefix** of P_{safe}, and we let $BadPref(P_{safe})$ denote the set of all bad prefixes of P_{safe}.
- $\hat{\sigma}$ is a **minimal bad prefix** for P_{safe} iff $\hat{\sigma} \in BadPref(P_{safe})$ and no proper prefix of $\hat{\sigma}$ is in $BadPref(P_{safe})$.
- Any invariant is a safety property. There are safety properties, that are not invariants.
- For a transition system TS without final states and safety property P_{safe} the following holds:

$$TS \models P_{safe} \text{ iff } Traces_{fin}(TS) \cap BadPref(P_{safe}) = \varnothing$$

- The **closure** of an LT property P is defined as:

$$closure(P) = \{\sigma \in (2^{AP})^{\omega} | pref(\sigma) \subseteq pref(P)\}$$

This denotes the set of all infinite traces whose finite prefixes are also prefixes of P Example: $closure(a^{+}b^{\omega}) = a^{\omega} + a^{+}b^{\omega}$

- P is a safety property iff $closure(P) = P$

- Finite traces and safety properties:
  - $Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$ iff for any safety property P_{safe} : $TS' \models P_{safe} \Rightarrow TS \models P_{safe}$
  - $Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$ iff TS and TS' satisfy the same set of safety properties
  - for TS without terminals states and finite TS' the following holds:

$$Traces(TS) \subseteq Traces(TS') \iff Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$$

  - trace inclusion $\neq$ finite trace inclusion

Liveness Properties

Safety properties can be satisfied by never doing anything. ⇒We want some property that ensures something will happen eventually.

An LT property P_{live} over AP is a **liveness** property whenever $pref(P_{live}) = (2^{AP})^{*}$

⇒Any finite computation can be extended to an execution satisfying the property.

Safety and Liveness component

Many properties are not pure, but a combination of safety and liveness properties.

- let S denote an atomic predicate that holds in a "sent" state, and let R denote an atomic predicate that holds in a "received" state.
- let $S, R \in \Sigma$ Σ is the alphabet
- Property: when a message has been "sent" eventually it will be "received":

$$\Pi = S^{*}R\Sigma^{\omega}$$

- $$\begin{aligned} \Pi &\neq closure(\Pi) \\ &= S^{\omega} \cup S^{*}R\sigma^{\omega} \\ &= \Pi_s \end{aligned}$$

  We call Π_s the **safety closure** of Π

- decomposition into safety and liveness property:

  liveness: Π_l eventually R will hold: $\Pi_l = \Sigma^{*}R\Sigma^{\omega}$
  safety: $\Pi_s = closure(\Pi)$
  will hold: $\Pi = \Pi_s \cap \Pi_l = (S^{\omega} \cup S^{*}R\Sigma^{\omega}) \cap (\Sigma^{*}R\Sigma^{\omega})$

- Safety and Liveness properties are **disjoint** except for the trivial property Σ^{ω}
- Every property Π can be represented as the **intersection**: $\Pi = \Pi_s \cap \Pi_l$
- let Π be a infinitary and Φ be an finitary property

- $PREF(\Pi)$ denotes the set of all finite prefixes of Π
- $A(\Phi)$ corresponds to all infinite $\sigma \in \Sigma^\omega$ so that all finite prefixes of σ are in Φ

$$\text{Example: } \Phi = a^+b^* \Rightarrow \underset{(closure)}{A(\Phi)} = a^\omega + a^+b^\omega$$

- $E(\Phi)$ consists of all infinite $\sigma \in \Sigma^\omega$ so that there exists a prefix of σ that belongs to Φ

$$\text{Example: } \Phi = a^+b^* \Rightarrow E(\Phi) = a^+b^* \cdot \Sigma^\omega$$

- Other Classifications:
  - topological ⇒safety (closed sets), liveness (dense sets)
    Note: for every finite property Φ : $E(\Phi) = \Phi \cdot \Sigma^\omega$
  - temporal logic
  - automata theoretic
- Safety/Liveness is intuitive:
  - checking safety properties: simple exploration of all states
  - checking liveness property: exploration of all states, checking in every state whether any continuation of prefix will satisfy property

Fairness

Fairness ensures, that each process wanting to do something at one point will eventually be able to do so.

Types of Fairness

unconditional fairness: an activity is executed **infinitely often**

strong fairness: if an activity is **infinitely often enabled**, it will **infinitely often be executed**

weak fairness if an activity is **continuously enabled**, it will **infinitely often be executed**

- Fairness violations often occur due to an unjustifiedly high level of abstraction (e.g. one process indefinitely faster than other)
- or due to unjustified assumptions about environment behaviour

For $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states, $A \subseteq Act$, and infinite execution fragment $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of TS where $Act(s) = \{\alpha \in Act | \exists s' \in S.s \xrightarrow{\alpha} s'\}$

1. ρ is **unconditionally A-fair** whenever: $true \Rightarrow \underbrace{\forall k \geq 0.\exists j \geq k.\alpha_j \in A}_{\text{infinitely often } A \text{ is taken}}$
2. ρ is **strongly A-fair** whenever: $\underbrace{(\forall k \geq 0.\exists j \geq k.Act(s_j) \cap A \neq \varnothing)}_{\text{infinitely often } A \text{ is enabled}} \Rightarrow \underbrace{\forall k \geq 0.\exists j \geq k.\alpha_j \in A}_{\text{infinitely often } A \text{ is taken}}$
3. ρ is **weakly A-fair** whenever: $\underbrace{(\exists k \geq 0.\forall j \geq k.Act(s_j) \cap A \neq \varnothing)}_{\text{infinitely often } A \text{ is enabled}} \Rightarrow \underbrace{\forall k \geq 0.\exists j \geq k.\alpha_j \in A}_{\text{infinitely often } A \text{ is taken}}$

Which Fairness Notion to Use?

Fairness filters out "unreasonable" runs (may be too strong, unfair)

- Too Strong? ⇒relevant computations may be ruled out, verification yields:

  false: error found
  true: may still be false, if relevant executions may be considered "unfair"
- Too Weak? ⇒too many computations considered, verification yields:

  true: property holds
  false: may still be true, if violating run was unfair
- Relationship: unconditional ⇒strong ⇒weak

Fairness Assumptions

A **fairness assumption** for Act is a triple

$$\mathcal{F} = (\mathcal{F}_{uncond}, \mathcal{F}_{strong}, \mathcal{F}_{weak})$$

with $\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak} \subseteq 2^{Act}$

- unconditionally A-fair **for all** $A \in \mathcal{F}_{ucond}$
- stongly A-fair **for all** $A \in \mathcal{F}_{strong}$
- weakyl A-fair **for all** $A \in \mathcal{F}_{weak}$

e.g. $(\varnothing, \mathcal{F}', \varnothing)$ denotes strong fairness

- A Path $s_o \to s_1 \to s_2 \to \dots$ is $\mathcal{F}$-fair, if
  - there exists an $\mathcal{F}$-fair execution $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$
  - $FairPaths_{\mathcal{F}}(s)$ denotes the set of $\mathcal{F}$-fair paths that start in s
  - $FairPaths\mathcal{F}(TS) = \bigcup_{s \in I} FairPaths\mathcal{F}(s)$
- Trace σ is $\mathcal{F}$-fair if there exists an $\mathcal{F}$-fair execution ρ with $trace(\rho) = \sigma$
  - $FairTraces_{\mathcal{F}}(s) = trace(FairPaths_{\mathcal{F}}(s))$
  - $FairTraces_{\mathcal{F}}(TS) = trace(FairPaths_{\mathcal{F}}(TS))$
- TS **satisfies** LT-property P:
$$TS \models P \text{ iff } Traces(TS) \subseteq P$$
- TS **fairly satisfies** LT-property P wrt. fairness assumption $\mathcal{F}$:
$$TS \models_{\mathcal{F}} P \text{ iff } FairTraces_{\mathcal{F}}(TS) \subseteq P$$

Fairness and Safety

- for TS with set of Actions Act and fairness assumption $\mathcal{F}$ for Act, $\mathcal{F}$ is defined as **realizable** for TS if for any $s \in Reach(TS)$:
$$FairPaths_{\mathcal{F}}(s) \neq \varnothing$$
- for TS and safety property P_{safe} over AP and $\mathcal{F}$ a realizable fairness assumption for TS the following holds true:
$$TS \models P_{safe} \text{ iff } TS \models_{\mathcal{F}} P_{safe}$$

i.e. safety properties are preserved by realizable fairness assumption, non-realizable fairness assumptions may harm safety properties

Model Checking Regular Safety Properties

A safety property P_{safe} is **regular** if

- $BadPref(P_{safe})$ is a regular language.
- there exists a finite automaton over 2^{AP} recognizing $BadPref(P_{safe})$

- $bp(P_{safe} \equiv BadPref(P_{safe})$
- $mbp(P_{safe})$ $\Rightarrow$set of minimal bad prefixes for P_{safe}
- not regular safety properties: something to do with counting

Regular Languages

A **nondeterministic finite automaton** NFA $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states
- Σ is an alphabet
- $\delta: \ Q \times \Sigma \to 2^Q$ is a transition function (nondeterministic)
- $Q_0 \subseteq Q$ a set of initial states
- $F \subseteq Q$ is a set of accept (or: final) states

assume word $w = A_1 \dots A_n \in \Sigma^*$

- A **run** for w in $\mathcal{A}$ is a finite sequence $q_0 q_1 \dots q_n$ such that:

$$q_0 \in Q_0 \text{ and } q_1 \xrightarrow{A_{i+1}} q_{i+q} \text{ for all } 0 \leq i < n$$

- Run $q_0 q_1 \dots q_n$ is **accepting** if $q_n \in F$
- $w \in \Sigma^*$ is **accepted** by $\mathcal{A}$ if there exists an accepting run for w.
- The **accepted language** of $\mathcal{A}$:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* | \text{ there exists an accepting run for } w \text{ in } \mathcal{A}\}$$

- NFA $\mathcal{A}$ and $\mathcal{A}'$ are **equivalent** if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$

Synchronous Product

For transition system $TS = (S, Act, \to, I, AP, L)$ without terminal states and $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and NFA with $\Sigma = 2^{AP}$ and $Q_0 \cap F = \varnothing$, let:

$$TS \otimes \mathcal{A} = (S', Act, \to', I', AP', L')$$

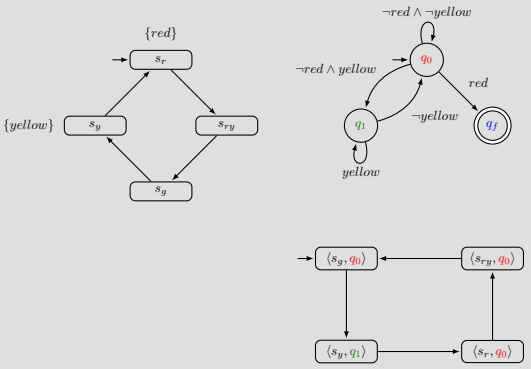
where

- $S' = S \times Q$, $AP' = Q$ and $L'(\langle s, q \rangle) = \{q\}$
- $\to'$ is the smallest relation defined by

$$\frac{s \xrightarrow{\alpha} t \wedge q \xrightarrow{L(t)} p}{\langle s, q \rangle \xrightarrow{\alpha}{}' \langle t, p \rangle}$$

- $I' = \{\langle s_0, q \rangle | s_0 \in I \wedge \exists q_0 \in Q_0 . q_0 \xrightarrow{L(s_0)} q\}$

without loss of generality it may be assumed that $TS \otimes \mathcal{A} has no terminal states$



Properties of NFA

- They are as expressive as regular languages
- They are closed unter $\cap$ and **complementation**
  - NFA $\mathcal{A} \otimes B$ (=cross product) accepts $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$
  - Total DFA $\overline{\mathcal{A}}$ (=swap all accept and normal states) accepts $\overline{\mathcal{L}(\mathcal{A})} = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$
- They are closed under **determinization** (=removal of choice) (although exponentially more runtime)

- $\mathcal{L}(\mathcal{A} = \varnothing \Rightarrow$check for a reachable accept state in $\mathcal{A} \Rightarrow$simple Dfs
- For regular language $\mathcal{L}$ there is a unique minimal Dfa accepting $\mathcal{L}$

Model Checking regular safety properties

- let P_{safe} be a regular safety property over AP
- let $\mathcal{A}$ be an Nfa recognizing the bad prefixes of P_{safe}
  - assume that $\varepsilon \notin \mathcal{L}(\mathcal{A})$ (otherwise all finite words over 2^{AP} are bad prefixes and $P_{safe} = \varnothing$)
- let TS be a finite transition system over AP without terminal states
- $TS \models P_{safe}$
  - iff $Traces_{fin}(TS) \cap bp(P_{safe}) = \varnothing$
  - iff $Traces_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$
  - iff $TS \otimes \mathcal{A} \models$ "always Φ"
    - invariant checking $\Rightarrow$Dfs

Procedure

1. Model transition system out of the model
2. Ltl needs to be created an negated
3. Create Nnf from this negated Ltl
4. Create Büchi Automaton from this Nnf
5. Synchronous product of TS and Büchi automaton needs to be created, Endstates are endstates of Büchi-automaton
6. conduct NDfs to find loops touch endstates in this system

The following statements are equivalent
- $TS \models P$
- $Traces_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$
- $TS \otimes \mathcal{A} \models P_{inv(\mathcal{A})} = \bigwedge_{q \in F} \neg q$

$\Rightarrow$checking safety properties is like checking an invariant

Counterexample

For each initial path fragment $\langle s_0, q_1\rangle \dots \langle s_n, q_{n+q}$ of $TS \otimes \mathcal{A}$:

$$q_1, \dots q_n \notin F \text{ and } q_{n+1} \in F \Rightarrow \underbrace{trace(s_0 s_1 \dots s_n) \in \mathcal{L}(\mathcal{A})}_{\text{bad prefix for } P_{safe}}$$

Input: finite transition system TS and regular safety property P_{safe}
Result: true if $TS \models P_{safe}$. Otherwise false plus a counterexample for P_{safe}
if $TS \otimes \mathcal{A} \models P_{inv(\mathcal{A})}$ **then**
  return true
else
  Determine initial path fragment $\langle s_0, q_1\rangle \dots \langle s_n, q_{n+1}\rangle$ of $TS \otimes \mathcal{A}$ with $q_{n+q} \in F$
  return (false, $s_0 s_1 \dots s_n$**)**
end

Algorithm 1: Basic Model Checking algorithm

Time Complexity

The time and space complexity of checking $TS \models P_{safe}$ is in:

$$\mathcal{O}(|TS| \cdot |\mathcal{A}|)$$

where $\mathcal{A}$ is an NFA with $\mathcal{L}(\mathcal{A} = mbp(P_{safe})$
The **size** of NFA $\mathcal{A}$, denoted $|\mathcal{A}|$ is the number of states and transitions in $\mathcal{A}$:

$$\underset{\text{rel. small}}{|\mathcal{A}|} = |Q| + \sum_{q \in Q} \sum_{A} \in \Sigma |\delta(q, A)|$$

Büchi Automata & ω-Regular Languages

- Properties of reacive systems contain liveness elements, need to be described with infinite sequences
- Need acceptance of languages of infinite words

Regular Expression

- Let Σ be an alphabet with $A \in \Sigma$
- Regular Expressions over Σ have the **syntax**:

$$E := \varnothing \mid \varepsilon \mid A \mid E + E' \mid E.E' \mid E^*$$

- The **semantics** of regular expression E is a language $\mathcal{L}(E) \subseteq \Sigma^*$:

$$\mathcal{L}(\varnothing) = \varnothing,\ \mathcal{L}(\varepsilon) = \{\varepsilon\},\ \mathcal{L}(A) = \{A\}$$
$$\mathcal{L}(E + E') = \mathcal{L}(E) \cup \mathcal{L}(E')\ \mathcal{L}(E.E') = \mathcal{L}(E).\mathcal{L}(E')\ \mathcal{L}(E^*) = \mathcal{L}(E)^*$$

ω-Regular Expressions

- denote languages of infinite words
- An ω-regular expression G over Σ has the form

$$G = E_1.F_1^\omega + \cdots + E_n.F_n^\omega \quad \text{for } n > 0$$

  where E_i, F_1 are regular expressions over Σ with $\varepsilon \notin \mathcal{L}(F_i)$
- The **semantics** of G is a language $\mathcal{L}(G) \subseteq \Sigma^\omega$:

$$\mathcal{L}_\omega(G) = \mathcal{L}(E_1).\mathcal{L}(F_1)^\omega \cup \cdots \cup \mathcal{L}(E_n).\mathcal{L}(F_n)^\omega$$

- G_1 and G_2 are **equivalent**, denotet $G_1 \equiv G_2$, if $\mathcal{L}_\omega(G_1) = \mathcal{L}_\omega(G_2)$
- $\mathcal{L}$ is ω-regular if $\mathcal{L} = \mathcal{L}\omega(G)$ for some ω-regular expression G.
- ω-Regular languages are closed under $\cup, \cap$ and **complementation**

ω-Regular Properties

An LT property P over AP is **ω-regular** if P is an ω-regular language over the alphabet 2^{AP}

Acceptance conditions for words inf $(2^{AP})^\omega$

1. finite acceptance (the automaton accepts a prefix of $a \in (2^{AP})^\omega$), does not ensure liveness, note even infiniteness

2. looping acceptance (the automaton has an infinite execution when reading the word) does not ensure liveness, as the automaton may do anything
3. repeating acceptance (looping acceptance + the set of states that the automaton reaches infinitely often an additional acceptance condition) also called "Büchi acceptance condition"

ω-Automata

- Definition as for NFA
- \+ Büchi acceptance condition: **An infinite word $a \in (2^{AP})^\omega$ will be accepted by a Büchi Automaton iff the automaton reaches at least one of the states in the acceptance set F infinitely often when reading a.**

A **nondeterministic Büchi automaton** NBA $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where:

- Q is a finite set of states with $Q_0 \subseteq!$ a set of initial states
- Σ is an alphabet
- $\delta: \ Q \times \Sigma \to 2^Q$ is a transition function (nondeterministic)
- $F \subseteq Q$ is a set of accept (or final) states

Language accepted by NBA

- NBA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and word $\sigma = A_0 A_1 A_2 \cdots \in \Sigma^\omega$
- A **run** for σ in $\mathcal{A}$ is an **in**finite sequence $q_0 q_1 q_2 \ldots$ such that:

$$q_0 \in Q_0 \text{ and } \underbrace{q_i \xrightarrow{A_{i+1}} q_{i+1}}_{\delta \text{ of the NBA}} \text{ for all } 0 \leq i$$

- Run $q_0 q_1 q_2 \ldots$ is **accepting** if $q_i \in F$ for **infinitely** many i
- $\sigma \in \Sigma^\omega$ is **accepted** by $\mathcal{A}$ if there exists an accepting run for σ
- The **accepted language** of $\mathcal{A}$:

$$\mathcal{L}_\omega(\mathcal{A}) = \{\sigma \in \Sigma^\omega | \text{ there exists an accepting run for } \sigma \text{ in } \mathcal{A}\}$$

- NBA $\mathcal{A}$ and $\mathcal{A}'$ are **equivalent** if $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}')$

More facts about Büchi Automata

- assume: the final state will be repeatet infinitely often
- finite equivalence $\not\Rightarrow$ ω-quivalence
- ω-equivalence $\not\Rightarrow$ finite equivalence
- The class of languages accepted by NBA is **identical** to the class of ω-regular languages
- A Büchi automaton is **deterministic**, if $|Q_0| \leq 1$ and $|\delta(q, A)| \leq 1$ for all $q \in Q$ and $A \in \Sigma$
  NBA and DBA are note equally expressive $\Rightarrow$NBA are more expressive $\Rightarrow$need for nondeterminism
- There is no canonic representation for NBA $\Rightarrow$there is no minimal NBA for a given ω-regular language
- alternative: "Muller, Rabin and Streett automata" with deterministic/nondeterministic variants $\Rightarrow$same expressivenes as N/DBA
- Generalized Nondeterministic Büchi Automaton GNBA:
  - acceptance condition expressed in terms of a set $\mathcal{F}$ consisting of sets $F_1, \ldots, F_k$ with $F_i \subseteq Q$
  - GNBA accepts a word, if it visits all sets F_i infinitely often

Linear Temporal Logic LTL

- Büchi automata use operational approach, want to have Logic for the description of temporal requirements
- first order logic quite complicated for this

$\Rightarrow$ need explicit quantification over state sequences

- modalities:
  Lp : it is necessary, that p — for all visible worlds p
  Mp : it is possible, that p — there exists a visible world with p
  $\neg Lp$: it is not necessary, that p
  $\neg Mp$: it is not possible, that p
- Formulae of Modal Logic:
  - p is a formula
  - if Φ is a formula, then $\neg\Phi$ is a formula
  - if Φ and ρ are formulae, then $\Phi \wedge \rho, \Phi \supset \rho$ (implication), $\Phi \equiv \rho, L\Phi, M\Phi$ are formulae

Let Π a set of atomic propositions. Further let

- W a set of worlds
- $R \subseteq W \times W$ a visibility relation on worlds
- $A : W \times \Pi \to \{true, false\}$ a truth value assignement

then we call (W, R, A) a **model** or **Kripke-Structure**

Model-theoretic Semantics of Modal Logic

Let

- $M = (W, R, A)$ a Kripke-structure
- Π a set of atomic propositions and $p \in \Pi$
- w and $v \in w$
- Φ and ρ formulae

then we define the relation $\models$ for M:

$$
\begin{aligned}
(M, w) &\models p && iff\ A(w, p) = true \\
(M, w) &\models \neg p && iff\ A(w, p) = false \\
(M, w) &\models \Phi \wedge \rho && iff\ (M, w) \models \Phi \text{ and } (M, w) \models \rho \\
(M, w) &\models L\Phi && iff\ (\forall v : (w, v) \in R)((M, v) \models \Phi) \\
(M, w) &\models M\Phi && iff\ (\exists v : (w, v) \in R)((M, v) \models \Phi)
\end{aligned}
$$

Further we define:

$$
\begin{aligned}
\Phi \vee \rho &\cong \neg(\neg\Phi \wedge \neg\rho) \\
\Phi \supset \rho &\cong \neg\Phi \vee \rho \\
\Phi \equiv \rho &\cong (\Phi \supset \rho) \wedge (\Phi \subset \rho) \\
M\Phi &\cong \neg L \neg \Phi
\end{aligned}
$$

Axiomatisation:

1. $(\Phi \vee \Phi) \supset \Phi$
2. $\rho \supset (\Phi \vee \rho)$
3. $(\Phi \vee \rho) \supset (\rho \supset \Phi)$
4. $(\Phi \supset \rho) \supset ((\rho \vee \Phi) \supset (\sigma \vee \rho))$
5. $L\Phi \supset \Phi$
6. $L(\Phi \supset \rho) \supset (L\Phi \supset L\rho)$
7. $L\Phi \supset LL\Phi$
8. $M\Phi \supset LM\Phi$

We call a formula Φ a **theorem** if it can be derived from the axioms, notation: $\vdash \Phi$

1. $\vdash \Phi$ and $\vdash (\Phi \supset \rho)$ imply $\vdash \rho$ (modus ponens)
2. $\vdash \Phi$ implies $\vdash L\Phi$

Temporal Interpretation of Modal Logic

- worlds correspond to system states
- visibility relation corresponds to sequences of system states
  - system sees its current state ⇒reflexivity
  - state s_3 follows s_2 and s_2 follows s_1, then s_3 follows s_1 ⇒transitivity
- linear time temporal logic Ltl:
  - reflexivity
  - transitivity
  - connectivity
  - discreteness

satisfiability and Validity

- p is **state-satisfiable** if there is a state so that $s \models p$
- p is **state-valid** if $s \models p$ for all states s

Linear Temporal Logic Ltl

p, q are formulae, the folowing are fomulae:

- $\circ p$ (next)
- $\diamond$ (eventually)
- $\Box$ (always)
- $p \mathcal{U} q$ (until)
- $p \mathcal{W} q$ (unless/weak until) either once q or always p

Satisfaction

given a state sequence (model) σ and formula p

- p **holds at position** i of sequence σ iff
$$(\sigma, i) \models p$$
we write $\sigma \models p$

Validity

a formula p is **valid**, iff
$$(\forall \sigma)(\sigma \models p)$$
we write $\models p$

Finite Model Property

LTL has the finite model property, there are only 4 distinct temporal modalities
$\Box p, \diamond p, \Box \diamond p, \diamond \Box p$
$\Box\Box \diamond p \equiv \Box \diamond p, \Box \diamond \Box \diamond p \equiv \Box \diamond p$

LTL-example

- $\Box(p \rightarrow \diamond q) \cong p \rightsquigarrow q$ "response-property", "p leads-to q"
- "between process A updating a value and process B reading the cache, the value must be flushed from A's cache"

$$\Box((UpdateA \wedge \diamond ReadB) \rightarrow (\neg ReadB \mathcal{U} FlushA))$$

Property Expansion

- $\Box p \Leftrightarrow (p \wedge \circ \neg p)$
- $\diamond p \Leftrightarrow (p \vee \circ \diamond p)$
- $p \mathcal{U} q \Leftrightarrow (q \vee [p \wedge \circ (p \mathcal{U} q)])$
- $p \mathcal{W} q \Leftrightarrow (q \vee [p \vee \circ (p \mathcal{W})])$

Dualitites

- $\neg \Box p \Leftrightarrow \diamond \neg p$
- $\neg \diamond p \Leftrightarrow \Box \neg p$
- $\neg (p \mathcal{U} q) \Leftrightarrow (\neg q) \mathcal{W} (\neg p \wedge \neg q)$
- $\neg (p \mathcal{W} q) \Leftrightarrow (\neg q) \mathcal{U} (\neg p \wedge \neg q)$
- $\neg \circ p \Leftrightarrow \circ \neg p$

Strong and Weak Operators

- $p \mathcal{U} q \Leftrightarrow (p \mathcal{W} q \wedge \diamond q)$
- $p \mathcal{W} q \Leftrightarrow (p \mathcal{U} q \wedge \Box q)$
- $p \mathcal{U} q \Rightarrow p \mathcal{W} q$

Indempotence

Two-fold application yields result identical to single application:

- $\Box\Box p \Leftrightarrow \Box p$
- $\diamond \diamond p \Leftrightarrow \diamond p$
- $p \mathcal{U} (p \mathcal{U} q) \Leftrightarrow p \mathcal{U} q$
- $p \mathcal{W} (p \mathcal{W} q) \Leftrightarrow p \mathcal{W} q$
- $(p \mathcal{U} q) \mathcal{U} q \Leftrightarrow p \mathcal{U} q$
- $(p \mathcal{W} q) \mathcal{W} q \Leftrightarrow p \mathcal{W} q$

Absorption

- $\diamond \Box \diamond p \Leftrightarrow \Box \diamond p$
- $\Box \diamond \Box p \Leftrightarrow \diamond \Box p$

Absorption and Indempotence also lead to only 4 different unary modalities (compare finite-model property):

$$\Box p, \diamond p, \Box \diamond p, \diamond \Box p$$

0.0.2 Commutativity of the next-operator

- $\circ(\neg p) \Leftrightarrow \neg \circ p$
- $\circ(p \vee q) \Leftrightarrow \circ p \vee \circ q$
- $\circ(p \mathcal{W} q) \Leftrightarrow \circ p \mathcal{W} \circ q$

Distribution of Temporal Operators

- $\Box(p \wedge q) \Leftrightarrow \Box p \wedge \Box q$
- $\diamond(p \vee q) \Leftrightarrow \diamond p \vee \diamond q$
- $\Box \diamond (p \vee q) \Leftrightarrow \Box \diamond p \vee \Box \diamond q$
- $\diamond \Box (p \wedge q) \Leftrightarrow \diamond \Box p \wedge \diamond \Box q$

Possible Base set of Operators

$\circ, \mathcal{W} \neg$, derived operators:

- $\Box p = p \mathcal{W} false$
- $\diamond p = \neg \Box \neg p = \neg (\neg p \mathcal{W} false)$
- $p \mathcal{U} q = p \mathcal{W} q \wedge \diamond q = p \mathcal{W} q \wedge \neg (\neg \mathcal{W} false)$

Possible Axiomatisation

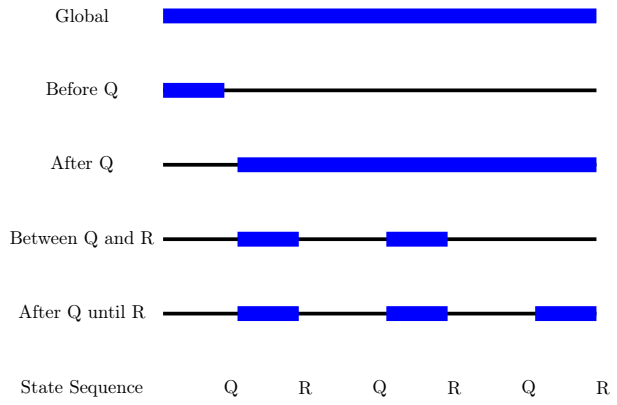
- $\Box p \rightarrow p$
- $\circ \neg p \Leftrightarrow \neg \circ p$
- $\circ(p \rightarrow q) \Leftrightarrow (\circ p \rightarrow \circ q)$
- $\Box(p \rightarrow q) \Leftrightarrow (\Box p \rightarrow \Box q)$
- $\Box p \rightarrow \Box \circ p$
- $(p \Rightarrow \circ p) \rightarrow (p \Rightarrow \Box p)$
- $p \mathcal{W} q \Leftrightarrow [q \vee (p \wedge \circ (p \mathcal{W} q))]$
- $\Box p \Rightarrow p \mathcal{W} false$

Safety-Progess Classification

- Temporal Logic Property Classes
  - classification based on syntactic form of formulae
  - A property $\Pi \subseteq \Sigma^\omega$ will be specified by temporal logic formula Φ if Φ is defined over the state vocabulary Σ and the following condition holds:

$$\sigma \in \Pi \quad \text{iff} \quad \sigma \models \Phi$$

  - hence, Φ describes a decision mechanism determining whether a state sequence belongs to the set of valid executions of a system
- alternative (Manna and Pnueli)
  - orthigonal to safety-liveness classification
  - classes:
    - safety $\Box p$
    - guarantee $\diamond p$
    - obligation $\diamond p \rightarrow \diamond q$ oder $\Box \vee \diamond q$
    - response $\Box \diamond p$
    - persistence $\diamond \Box p$
    - reactivity $\Box \diamond p \vee \diamond \Box q$
  - Temporal Logic Specification Patterns: `http://www.cis.ksu.edu/santos/spec-patterns/index.html`
  - Scopes used in Specification Patterns



Büchi Automata and LTL

Relationship of Büchi-Automata and LTL

- Büchi-Automata more expressive.
  - LTL can not count
ETL	Not counter-free Büchi-Automata	$\approx$	ω-regular languages
LTL	Counter-free Büchi-Automata	$\approx$	$^* - free\omega$-regular languages

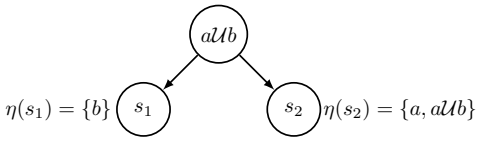
LTL to Büchi-Automaton Translation

- LTL often more abstract / easier than to directly specify as Büchi-Automata
- next-operator free LTL specified properties are guaranteed to be stutter invariant $\Rightarrow$Partial Order Reduction

Overview

- synthesize an automaton that uses Büchi acceptance criteria to represent the same set of models that satisfy the LTL formula

- let every node s of this automaton be labeled with an LTL formula $\eta(s)$
  - for some accepting run σ, if the automaton reaches state s, then the suffice $sigma[s\dots]$ must satisfy $\eta(s)$, i.e., $\sigma[s\dots] \models \eta(s)$
  - the $\eta(s)$ are of the form $(\wedge_{i=1\dots m}\nu_i \wedge \circ(\wedge_{j=1\dots n}\kappa_j)$, i.e. the successor nodes of s have to satisfy $(\wedge_{j=1\dots n}\kappa_j)$
  - refine $\eta(s)$ into shorter subformulas ν_i until all ν_i are propositional variables in positive or negative form



  - for $\Phi\mathcal{U}\Psi$
    - either satisfy Ψ now or Φ now and $\Phi\mathcal{U}\Psi$ later
    - split current node
      - first resulting node, add Ψ to ν_i
      - second resulting node, add Φ to ν_i and $\Phi\mathcal{U}\Psi$ to κ_i
    - keep all ν_i and κ_i formulas that are not affected by current splitting
    - when all formulas have been split, generate successor node s' to s
    - ν_i of s' are the κ_i of s
    - κ_i of s' are initially empty
    - no more splitting? define acceptance condition
  - Not necessarily the most effective algorithm, but no proof for more effective exists
  - steps of translation
    1. preprocessing: bring formulae into negation normal form
    2. construction of node data structure for Büchi Automaton: recursively decompose
    3. define acceptance criteria to turn node data structure into Büchi Automaton: Use acceptance criterion for GNBA
    4. translate the GNBA into a "simple" NBA
  - convert into **negation normal form**, where negation only applies to propositional variables

$$
\begin{aligned}
\neg\Box\Phi &\Rightarrow \Diamond\neg\Phi \\
\neg\Diamond\Phi &\Rightarrow \Box\neg\Phi \\
\neg(\Phi\mathcal{U}\Psi) &\Rightarrow (\neg\Phi)\mathcal{V}(\neg\Psi) \\
\neg(\Phi\mathcal{V}\Psi) &\Rightarrow (\neg\Phi)\mathcal{U}(\neg\Psi) \\
\neg(\Phi\vee\Psi) &\Rightarrow (\neg\Phi)\wedge(\neg\Psi) \\
\neg(\Phi\wedge\Psi) &\Rightarrow (\neg\Phi)\vee(\neg\Psi) \\
\Phi\rightarrow\Psi &\Rightarrow (\neg\Phi)\vee\Psi \\
\Diamond\Phi &\Rightarrow true\mathcal{U}\Phi \\
\Box &\Rightarrow false\mathcal{V}\Phi
\end{aligned}
$$

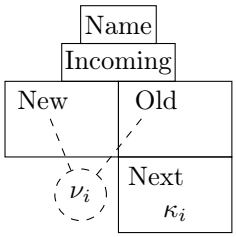
  - Semantics of $\mathcal{V}$ (unless): Φ is false when Ψ is true Is this true?
  - node data structure

    Name unique identifier for the node

    Incoming list of node names that lead into this node

    New,Old,Next each is a set of subformulas for Φ
    - each node represents suffixes of executions
    - $New(s)\,\sup\,Old(s)$: ν_i formulas
    - $New(s)$: unprocessed formulas
    - $Old(s)$: processed formulas
    - $Next(s)$: κ_i formulas

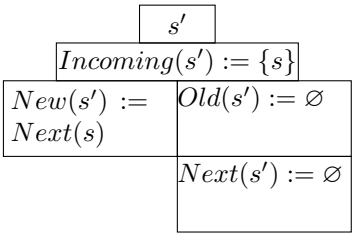
    Nodes_Set set of all existant nodes



Observation * assumption: $\sigma[\dots s]$ satisfies all subformulae in Old or New
* obligation: $\sigma[(s+1)\dots]$ satisfies all subformulae in Next

- Nodes can be split (two nodes after) or evolved (one node after)
- Processing of Current node s: is there a subformula to be processed in New of node s?

  no: node s is completely processed
  is there another node r with the same subformulae as s in both Old and $Next$ field? $\Rightarrow s$ is discarded and incoming edges of s are added to r
  else: add node s to $Nodes\_Set$, continue with new subnode s'



  yes: select subformula η from New
  * remove η from $New(S) := New(s) - \{\eta\}$
  * determine main Boolean operator in η $\Rightarrow$determine which rule to apply
  * perform split into s_1, s_2 or evolve to s'
    · if η is $\Phi \wedge \Psi$,**proposition, negated proposition** or **Boolean constant**:
    if η is $false$ or if $\neg\eta$ is in old $\Rightarrow$contradiction, discard
    else: s evolves into s'
    · if $\eta = \Phi\mathcal{U}\Psi,\ \Phi\mathcal{V}\Psi,\ \Phi \vee \Psi$, split into s_1, s_2

evolve → node gets changed only?

- Automaton construction form node structure
  - initial node is initial state
  - make transitions to states from all states in $Incoming$
  - GNBA Acceptance Conditions
    * create set F_i for each subformula of the form $\Phi\mathcal{U}\Psi$
    * this ensures that if $\Phi\mathcal{U}\Psi$ holds in some run, Ψ must hold later
    * F_i contains all states that either:
      · contain Ψ in its label
      · does not contain $\Phi\mathcal{U}\Psi$ in its label
- Translation from GNBA $\mathcal{B} = (\Sigma, S, \delta, I, L, F)$ to NBA $\mathcal{A}' = (\Sigma, S', \delta', I', F')$, $F = \{F_1 \dots F_n\}$
  - $S' = S \times \{1 \dots n\}$
  - $I' = I \times \{1\}$
  - δ' is defined by $(t, i) \in \delta'((s, k), a)$ if $t \in \delta(s, a)$ and

$$i = k if\ s \notin F_k$$
$$i = (k \mod n) + 1 if\ s \in F_k$$

  - $F' = F_1 \times \{1\}$
- the construction labels states with numbers from $1 \dots n$
  - if the current number is k, then it will keep k if we do not go though an acceptance state of F_k
  - if we do go though an acceptance state F_k, then the new value is $\underbrace{(k \mod n) + 1}_{\text{the next} k}$
  - As soon as we reach an acceptance set from F_1 we look for one from F_2 next and thus go through all acceptance sets

Complexity of Algorithm

- some optimization possible (redundancies, bisimulation reduction, weak alternating automata, prune parts that don't lead to acceptance cycle)
- number of nodes and runtime is exponential in the length of the Ltl formula
- Ltl are typically small ⇒Büchi automaton typically less than 10 states
- Ltl model checking is polynomial in the size of the Büchi automaton

Algorithms for Checking Safety Properties

Basic Dfs

```
Start() begin
    // s_0 is initial state of TS
    Add_Statespace(P.T.s_0)
    Push_Stack(P.T.s_0)
    Search()
end

Search() begin
    s=Top_Stack()
    // this checks the property and prints the counterexample if violated
    if !Safety(s) then
        Print_Stack(D)
    end
    // P.T.E ⇒ successors(s)
    forall the (s, s') ∈ P.T.E do
        // state not yet visited (uses hashtable)
        if In_statespace(s'( == false)) then
            Add_Statespace(s')
            Push_Stack(s')
            Search()
        end
        Pop_Stack()
    end
end
```

Algorithm 2: Basic Depth First Search

- properties for finite state spaces
  - complete exploration of state space
  - termination can be proven
- complexity
  - linear in the number of nodes (which can be exponential in number of components)
- efficiency problem
  - ever node needs to be visited twice, once during generation of asynchronous product Lts, once during state space search ⇒requires storage of copmlete $_{\text{reachable}}$ state space
  - on-the-fly algorithm ⇒expand asynchronous product and search

On-the-fly Dfs

```
Start() begin
    // s_0 is initial state of TS
    Add_Statespace(z_0)
    Push_Stack(z_0)
    Search()
end

Search() begin
    z=Top_Stack()
    // all successor states in all components
    forall the i with i ≤ i ≤ n do
        s = Control(z, i)
        v = Data(z)
        // all successor states in component i
        forall the (s, s') ∈ P.T.E do
            if F(s, s').C(v) == true then
                z' = Update(z, i, s', F(s, s').A(V)) if In_Statespace(z') == false then
                    Add_Statespace(z')
                    Push_Stack(z')
                    Search()
                end
            end
        end
    end
    Pop_Stack()
end
```

Algorithm 3: On-the-fly Depth First Search

- $Control(z, i)$: return control state of $i - th$ component
- $Data(z)$: return data values $v_1, \ldots, v_n$
- $Update(z, i, s, f)$: take z, replace control of $i - th$ component with s, apply data action f ⇒check property

Basic-Dfs vs. On-the-fly Dfs

- Trade space complexity (memory) for time-complexity
  - On-the-fly Dfs erases the parts of the state space **not on the stack**
  - danger of multiple generation of some states if erased and need to be visited again
- in practical situations: Size (exponential in ...) prevents use of basic dfs algorithm

Bfs

```
Queue D = {}
Statespace V = {}
Start() begin
    // s_0 is initial state of TS
    Add_Statespace(V, A.s_0)
    Add_Queue(D, A.s_0)
    Search()
end

Search() begin
    z=Del_Queue()
    foreach (s, l, s') ∈ A.T do
        if In_Statespace(V, s') == false then
            Add_Statespace(V, s')
            Add_Queue(D, s')
            Search()
        end
    end
end
```

Algorithm 4: Basic Breatdh-First Search

Depth-First vs. Breadth-First Search

BFS always finds shortest counterexample

otf-DFS
- easy extension to detect cylces (liveness properties)
- stack contents automatically yield counterexample
- memory efficiency, only need to store portion of state space

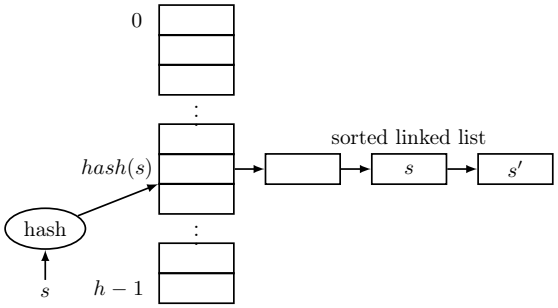
conclusion Dfs will be used in most cases, moderately sized models with Bfs

Spin Depth-Bounded Breadth First Search

- May lead to incomplete search

0.1 Bitstate Hashing (Supertrace)

Don't want to revisit states, that have previously been visited. To efficiently determine, whether a state was visited before: use a **hashtable**



- h slots with lists of zero or more states
  - store new state s: compute $hash(s)$ out of $0 \ldots (h-1)$
  - lookup state s: compute $hash(s)$, search for s in sorted linked list, if not found add s to linked list
  - Spin uses checksum-polynomial as well as Jenkin's hashing
- impact of hash collisions:
  - r number of states stored in hash table
  - h number of slots in table
  - if $h >> r$ then each state can be stored in different slot for equal distribution
    - $\Rightarrow$ linked list in every slot either empty or just one element
    - $\Rightarrow$ almost no storage and time overhead

- if $h < r$, then hash collisions lead to lists with $length \geq 1$
  - $\Rightarrow$ on average $\frac{r}{h}$ comparisions in linked list required per state access
  - $\Rightarrow$ overhead grows linearly with $\frac{r}{h}$
- if $h >> r$: what should we do
  - store just **one bit**, indicating whether state was visited before, or not
  - dramatic savings in state space consumption
  - problem of duplicates:
    - renders method incomplete, when $hash(s) = hash(s')$ for $s \neq s'$ (two different states with same hash, algo will assume, that state was visited before)
    - does not compromise soundness of method (every error, that is found is indeed an error, not all errors may be found)
- increased coverage due to bistate hashing
  - m : bit to store in hash table
  - S : bits per state description
  - r : number of reachable states
  - h : slots in hash table
- less than $\frac{m}{s}$ will fit in memory, since hash table itsself requires space
- if $r > \frac{m}{S}$, search will abort after exploring a fraction of $\frac{m}{r*S}$ of the state space
- practical experience shows, that bitstate hashing produces greater coverage due to ability to store more states

Model Checking ω-Regular Properties

To validate liveness-properties we need to visit every state and check, whether a cycle violating the condition can be found. This can be achieved by two nested Dfs'

Persistence

We call **"eventually forever $\neg F$"** a **persistence** property:
Let $P_{pers} \subseteq (2^{AP})^{\omega}$ an LT property and Φ some propositional logic formula over AP. We call

$$P_{pers} = \{A_0 A_1 A_2 \cdots \in (2^{AP})^{\omega} | \exists i \geq 0. \forall j \geq i. A_j \models \Phi\}$$

a **persistence property** and Φ a **persistence** (or state) **condition** of P_{pers}. Φ represents a propositional property that after a while (=a finite number of steps) becomes an invariant

Persistence Checking

- Check wheter $TS \not\models P_{pers}$ with persistence condition Φ
- Algorithmic Idea
  - Let s a state reachable in TS with $S \not\models \Phi$ $\Rightarrow$then TS has an initial path fragment leading into s
  - If s is on a cycle
    - $\Rightarrow$ this path fragment can be continued by an infinite path in TS by re-visiting s infinitely often
    - $\Rightarrow$ TS may visit the $\neg\Phi$-state s infinitely often, which hence means

$$TS \not\models P_{pers}$$

  - If no $\neg\Phi$-state s on a cycle can be found, then $TS \models P_{pers}$

$TS \not\models P_{pers}$ iff $\exists s \in Reach(TS).s \not\models \Phi \wedge s$ is on a cycle in $G(TS)$
The following statements are equivalent

1. $TS \not\models P$
2. $Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$
3. $TS \otimes \mathcal{A} \models P_{pers(\mathcal{A})}$

- model checking ω-regular properties is reduced to checking persistence properties and thus finding cycles in $TS \otimes \mathcal{A}$

Nested Depth First Search NDFS

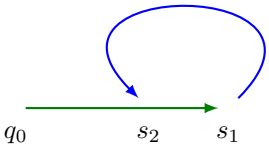
performs two nested DFS searches, post-order traversal for the first DFS.

```
// emptiness of L(TS ⊗ A)
emptiness() begin
  dfs1(q0)
  terminate(false)
end

dfs1(q) begin
  local q'
  // mark q as visited by dfs1
  hash(q)
  // on-the-fly
  forall the successors q' of q do
    if q' not in hashtable then
      dfs1(q')
    end
  end
  // acceptance state
  if q ∈ F then
    dfs2(q)
  end
end

// All states visited by dfs2 have previously been generated by dfs1
dfs2(q) begin
  local(q')
  // mark q as visited by dfs2
  flag(q)
  forall the successors q' of q do
    if q' on dfs1-stack then
      terminate(true)
    else
      if q' not flagged then
        dfs2(q')
      end
    end
  end
end
```

Algorithm 5: Nested Depth First Search

Derivation of counterexample for `terminate(true)`

- dfs2(s_1)
- s_2 is found by dfs2 on dfs1-stack
- counterexample construction: $(q_0 \dots s_2)(s_2 \dots s_2)^\omega$



Observations

- NDFS will not produce all accepting runs, but just one, iff there is one (i.e. completeness is not compromised)
- DFS2 will only encounter states that have previously been reached by DFS1 (post-order traversal) ⇒a state, once flagged will not be re-expanded during cycle detection
- Post-Order vs. Pre-Order Traversal Strategy
  - alternative: pre-ordered nested search
    - search an acceptance state and once encountered, find a cycle
    - quadratic worst case time overhead, linear worst case space overhead ⇒since second search has to be invoked with a newly initialized list of flagged states
    - ⇒ post-order traversal strategy of NDFS is more efficient

Nested DFS vs. Tarjans's SCC Detection

Tarjan

- SCC: strongly connected component: each element is reachable from inside an SCC
- Tarjan: DFS-based SCC detection (not on-the-fly)
- every state is visited twice
- memory-overhead: thee $32 - bit$ numbers / addresses per state visited
- capable to find all accepting paths
- easier to implement strong fairness constraints

NDFS

- same worst case complexity
- memory overhead: $2 - bits$ per state (hash + flag)
- enables supertrace / bitstate hashing
- finds one accepting path, if one exists (suffices to refute claimed property)

Property Specification in Promela

- Progress State Labels
  - processes may diverge in infinite loops without ever making any visible progress
  - prepending "progress" in front of a name of a control state label means, that any cyclic execution of the Promela model must lead though at least on progress labeled state, otherwise "non-progress-cycle"
- Temporal or Never Claims
  - possibility to specify temporal properties directly as Büchi automata
  - Promela model and never claim are executed as synchronous product
- also permitted: end, progress, assert
- LTL gets translated to never claim, as those are often unintuitive

Path Properties

- evaluated along execution paths
- Let R an asynchronous product of a set of MTS

- R is **free of non-progress cycles** if at least one state in the cyclic suffix of any cyclic trace of $R.P.T.E$ in $R.PRG$ (progress states)
- R is **free of accepting cycles** if no state of the cyclic suffix of any cyclic trace of $R.P.T.E$ is in $R.ACC$ (acceptance states F)

- need for cycle-searching algorithm for accepting cycle detection
  - traverse state space to find an accepting state x ⇒a procut state z is accepting if at least oine of the component MTS is in $M_i.ACC$
  - change to second state space, set x as seed, check whether x is reachable form itself ⇒post-order traversal

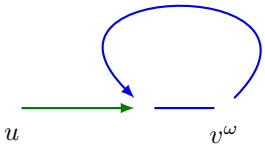
  ⇒NDFS
- on-the-fly Non-Progress Cycle Detection
  let $x \in M_i.PRG$ for all i
  - check, whether x is reachable from itself
  - if so, and if for all states y in this cycle $y \notin M_i.PRG$ for all i, then a non-progress cycle has been detected

Automata-based Model Checking

- Computing $L(M) \subseteq L(S)$
- let $\Sigma^\omega - L(S) = \overline{L(S)}$ then

$$L(M) \subseteq L(S) \leftrightarrow L(M) \cap \overline{(L(S))} = \varnothing$$

- implementation variants:
  - direct complementation of S: non-trivial operation
    for NBA S with n states, complement has $2^{\mathcal{O}(n \log n)}$ states
  - direct specification of $\overline{S}$
  - specify LTL formula φ, obtain $\neg\varphi$, tranlaste $\neg\varphi$ into equivalent NBA
- Existence of counterexample
  - $L(M) \cap \overline{L(S)} = \varnothing$ ⇒A satisfies S
  - $L(M) \cap \overline{L(S)} \neq \varnothing$ ⇒C is counterexample for the non-satisfaction of S by M
  - it can be shown, that any word in C can be represented by an ω-regular expression of the form uv^ω where u and v are finite state event sequences



- Construction of the Intersection of Büchi Automata
  - let $M_1 = (Q_1, q_0, q_{0,1}, A, \delta_1, F_1)$ and $M_2 = (Q_2, q_2, q_{0,2}, A, \delta_2, F_2)$ be two Büchi automata
  - the Büchi automaton accepting $L(M_1) \cap L(M_2)$ can be defined as

$$M_1 \cap M_2 = (Q_1 \times Q_2 \times \{0,1,2\}, (q_{0,1}, q_{0,2}, 0), A, \delta, Q_1 \times Q_2 \times \{2\})$$

  such that $(\langle r_i, q_i, x\rangle, a, \langle r_k, q_n, y\rangle) \in \delta$ iff all of the following contidions hold:

1. $(r_i, a, r_k) \in \delta_1$ and $(q_j, a, q_n) \in \delta_2$ (the transition of the intersection automaton agree with the transition of the operand automata)
2. The third component of the state tuples can be computed as follows:
   * if $x = 0$ and $r_k \in F_1$, then $y = 1$
   * if $x = 1$ and $q_n \in F_2$, then $y = 2$
   * if $x = 2$ then $y = 0$
   * else, $y = x$

– third component in state ensures, that acceptance states of both operand automata M_1 and M_2 occur infinitely often in accepting cycle

– construction is generally considered to inefficient

Emptiness Check for Büchi Automata

- Let $M = (Q, q_0, A, \delta, F)$ a Büchi automaton, let ρ a run on σ such that σ is being accepted by M
- ρ contains infinitely many accepting states from F
- since Q is finite, there is a suffix ρ' of ρ such that every state in ρ' occurs infinitely often
- Every state in ρ' is reachable from every other state in $\rho' \Rightarrow$states in ρ' form an Scc

Promela Never Claim

- just one instance per Promela model
- executed synchronously with the rest of the model
- can perform a step when condition label on transition is satisfied in current state of Promela model
- the final state of never claim is always indicating a property violation: stuttering semantics: final state repeated forever
- there is something missing here: part 15.44-47

Fairness

strong fairness: a ω-run σ satisfies the property of **strong fairness** if it contains infinitely many transitions from every componen automaton (proctype), which is infinitely often enabled in σ

$$\forall \text{ component automata } i : \Box \diamond enabled(i) \Rightarrow \Box \diamond taken(i)$$

weak fairness: a ω-run σ satisfies the property of **weak fairness** if it contains infinitely many transitions from every componen automaton (proctype), which is enabled infinitely long in σ

$$\forall \text{ component automata } i : \diamond\Box enabled(i) \Rightarrow \Box \diamond taken(i)$$

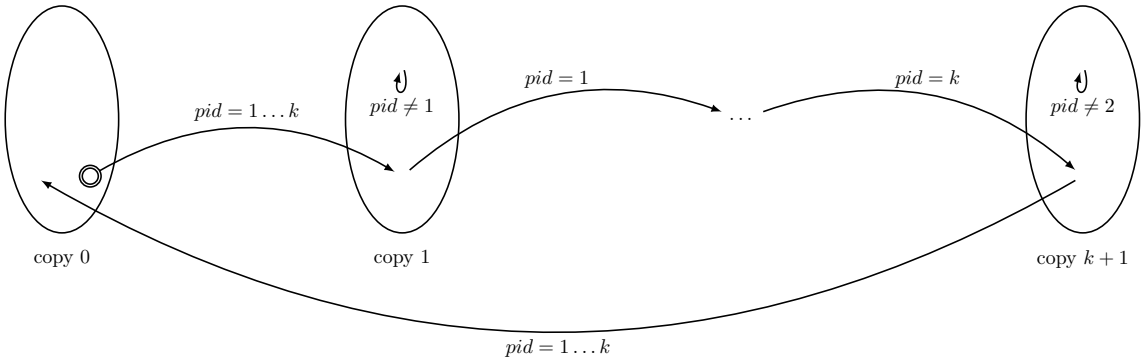
- avoidance of explicit specification of fairness constraints using Ltl formulae by **built-in** fairness checking algorithms

  strong fairness: runtime increase by factor n^2, where n is the number of concurrent components

  weak fairness runtime increase by factor n
  implemented in Spin
  in practice the factor is often 2

- Construction: based on Chouekas flag construction
- NDfs checks LTS consisting of k component LTSs
- create $k + 2$ copies of the LTS that is obtained from the asynchronous product of the component LTSs
- preserve acceptance labels only in copy 0

- change outgoing transitions from acceptance states in copy 0 to point to corresponding state in copy 1
- in ith component ($1 \leq i \leq k$) component, destination state of each transition contributed by LTS_i to corresponding state in LTS_{i+1}, transitions contributed by other LTSs remain in same component
- all outgoing transition in copy $k+1$ are pointing to copy 0
- a component LTS that has no enabled transition in some state s does not need to participate in infinite run that traverses s
- add a null transition from every state s in copy i to the same state S in copy i



- effect:
  - no change in behaviour
  - accepting ω run now contains transitions from all k component LTSs
  - there can be no accepting cycles in copy 0
- use NDFS on undolded graph to detect weakly fair acceptance cycles
- Implementation optimizations:
  - do not store $n+2$ copies, but a $k - bit$ vector indicating in which copy this state has been reached
- wirst case time penalty $2(k+2)$
  in practice: factor = 2

Partial Oder Reduction

for concurrent events $a \| b$ any possible interleaving will be part of set of valid system executions that the model checkers explores. Partial Order Reduction tries to prune execution sequences, that don't change the effect with respect to the property to be checked (i.e. reduces redundancy w.r.t. property).

```
explore_statespace() begin
    set on_stack(q_0)
    dfs(q_0)
end
dfs() begin
    work_set(q) := ample(q)
    local q'
    hash(q)
    forall the elements q' not in hashtable do
        on_stack(q')
        dfs(q')
    end
    set completed(q)
end
```

Algorithm 6: On-the-fly Depth-First-Search with Partial Order Reduction

Ample Set

Calculation of $ample(q) \subseteq enabled(q)$

- include sufficiently many elements from $enabled(q)$ so that model checking algorithm delivers **correct results**
- use of $ample(q)$ should lead to a **significantly smaller** state graph (in terms of states and transitions)
- computation of $ample(q)$ should be doable with **acceptable computation overhead**

A **state transition system** is a tuple (S, T, S_0, L) where

- S : finite set of states
- $S_0 \subseteq S$: finite set of initial states
- $L : S \to 2^{AP}$: function that labels every state with the AP true int that state
- T : finite set of transition relations so that for each $\alpha \in T, \alpha \subseteq S \times S$

let $\alpha \in T, s \in S$ then

- $\alpha \in enabled(s)$ iff $(\exists s' \in S)((s, s') \in \alpha)$
- α is deterministic if for every s, there is at most one s' so that $(s, s') \in \alpha$ (consider only deterministic transitions)
- we write $s' = \alpha(s)$ for $(s, s') \in \alpha$

Independence

$I \subseteq T \times T$ is an independence relation if I is symmetric and antireflexive and the following conditions hold for each $s \in S$ and for each $(\alpha, \beta) \in I$:

1. if $\alpha, \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$
   enabledness: a pair of independent transitions do not disable each other when taken
2. if $\alpha, \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$
   commutativity: executing pair of independent transitions in any order result in same state

Dependence Relation: $D := (T \times T) - I$

Correctness of Pruning

Elimination may still deliver incorrect results (even if you show independence before) if:

1. s_1 and s_2 may influence the outcome of the property check (the property is not insensitive to whether s_1 or s_2 is being reached)
2. s_1 or s_2 may have successor states other than r that would be pruned in case either of the two states did not belong to the reduced state space

0.2 Invisibility

$T = (S, T, S_0, L)$ a state transition system, $AP' \subseteq AP$
$\alpha \in T$ is **invisible** w.r.t AP' if

$$(\forall s, s' \in S | s' = \alpha(s))((L(s) \cap AP') = (L(s') \cap AP'))$$

(i.e. a transition is invisible w.r.t. some selected set of propositions if its execution doesn't change the truth value for the selected set of propositions)

0.3 Invariance under Stuttering

two paths σ and ρ though a state transition system are **stuttering equivalent** (written as $\sigma \sim_{st} \rho$) if the following condition holds:

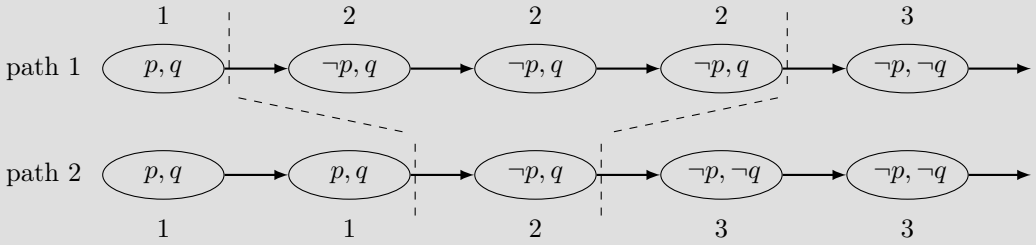
- there are two infinite sequences of integers

$$0 = i_0 < i_2 < \dots$$
$$0 = j_0 < j_1 < \dots$$

  such that for every $k \geq 0$

$$L(s_{i_k}) = L(s_{i_{k+1}}) = L(s_{k+1-1}) = L(r_{j_k}) = L(r_{j_{k+1}}) = L(r_{j_{k+1}-1})$$

- identically labeled sequences of states are called blocks



Invariance under Stuttering

An LTL formula f is invariant under stuttering, if for each pair of paths π and π' such that $\pi \sim_{st} \pi'$:

$$\pi \models f \text{ iff } \pi' \models f$$

- let LTL_χ denote the set of all LTL formulae free of the nexttime operator $\circ$
- any property expressible inLTL_χ is invariant under stuttering
- any stutter invariant property that is expressible with LTL can be expressed in LTL_χ

Stutter Invariance for Transition Systems
Let M and M' be state transition systems. M and M' are **stutter invariant** iff

- they have the same set of initial states
- for each path σ of M that starts in an initial state of M there is a path σ' of M' that starts in an initial state of M' such that $\sigma \sim_{st} \sigma'$
- the same for the other way around
- Let M and M' two stuttering equivalent state transition systems. For ever property expressed by an LTL_χ formula f and every initial state $s \in S_0$ the following holds true:

$$(M, s) \models f \text{ iff } (M', s) \models f$$

(LTL_χ formulae cannot distinguish between stuttering equivalent state transition systems)

Partial Order Reduction for LTL_χ

- s is fully expanded, iff $enabled(s) = ample(s)$
- else: provide conditions for selecting $ample(s)$ such that reduced state space satisfies property expressed by LTL_χ formula f

  C0: at-leat-one-successor rule

  $$(\forall s \in S)(ample(s) = \varnothing \text{ iff } enabled(s) = \varnothing)$$

  if a state has at least one successor in the full state space, it has at least on successor in the reduced state space

  C1: dependent-transition rule
  Along every path in the full state space that starts at s, a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occuring first.
  for all paths in the full state space starting at s, the following holds true:
  need to check for $C1$ without actually computing full state space in order to enable on-the-fly model checking, will later be done by approximation
  - a transition α' that is dependent on a transition $\alpha \in ample(s)$ cannot be executed without a transition from $ample(s)$ occuring first
  - the transition in $enabled(s) - ample(s)$ are all independent of those in $ample(s)$

  C2: invisibility rule
  for all states $s \in S$: if s is not fully expanded, then every $\alpha \in ample(s)$ is invisible

  C3: cycle condition (also called "Cycle Proviso")
  The reduced state graph may not contain a cycle in which $\alpha \in enabled(s)$ for some state s of the cycle so that $\alpha \notin ample(s')$ for all states s' of the cycle

- for LTL_χ property f, reduction depends on the set AP_f of atomic propositions occurring in f

Preservation of Correctness

C1: dependend transition rule

Ensure, that the DFS algorithm, when applied to the reduced statespace does not prune any parts of the graph, that are essential to the property when choosing the next transition to explore from $ample(s)$
$C1$ ensures that:

case 1: $\beta_0\beta_1 \dots \beta_m\alpha$, where $\alpha \in ample(s)$ and each β_i is independent of all transitions in $ample(s)$, including α

case 2: $\beta_0\beta_1 \dots$ where β_i is independent of all transitions in $ample(s)$

$\Rightarrow$ β_i is independent from transitions on $ample(s)$ $\Rightarrow$cannot disable them

C2: Invisibility rule

as alls transitions in $ample(s)$ are invisible, for $\alpha \in ample(s)$:

$$\alpha\beta_0\beta_1 \dots \text{ is stuttering equivalent to } \beta_0\beta_1$$

Partial Order Reduction and Search

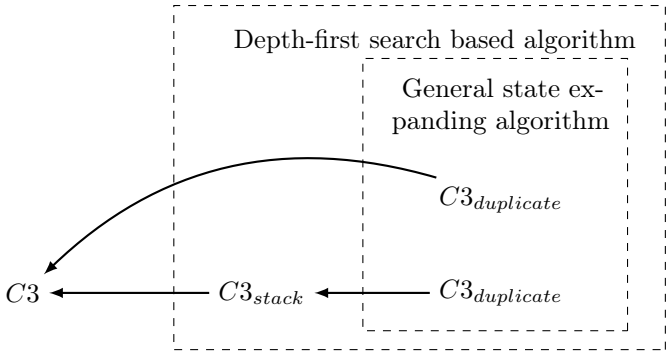
Conditions $C0, C1, C2$ are independent of search algorithm, $C3$ is global condition and as expensive to check as model checking itself

- $C3$ can be over-approximated as $C3_{cycle}$:
  - every cycle in the reduced state space contains at least one state, that is fully expanded ($enabled(s) = ample(s)$)
  - expensive, since it requires cycle detection

  - not on-the-fly
- $C3$ further over-approximated for safety-properties as $C3_{stack}$
  - if a state is not fully expanded, then at least one transition in $ample(s)$ does not lead to a state that is on the search stack
  - only applicable to stack-based DFS search
  - stronger than $C3$, less reduction
- Overapproximation of $C3$ for non-DFS (e.g. BFS, A^*)

$C3_{duplicate}$: if a state is not entirely expanded, then at least one transition in $ample(s)$ does not lead into a previously visited state ⇒will not necessarily close a cycle

$C3_{static}$: if a state is not fully expanded, then there is no transition in $ample(s)$ which closes a cycle in the control flow of a local process



Computational effort:

C0 easy, constant time

C1 checking this is at least as hard as checking reachability for full state space ⇒over-approximation heuristics

C2 easy, static analysis of the transitions in the set

Heuristics for C1

- $pc_i(s)$: program counter for process P_i in state s
- $pre(\alpha)$: all transitions β, such that there is a state s for which $\alpha \notin enabled(s), \beta \in enabled(s)$ and $\alpha \in enabled(\beta(s))$
- $dep(\alpha) = \{\beta | (\beta, \alpha) \in D\}$, the set of all dpeendent transitions of α
- $T_i(s)$: transitions of process i enabled in state s
- $current(s)$: set of transitions of P_i that are enabled in some state s' such that $pc_i(s') = pc_i(s)$

Definition of $pre(\alpha)$ and D may not be exact

$pre(\alpha)$ may contain transitions that do not enable α

D may contain transitions that are actually independent

This allows for efficient computation of ample sets while preserving correctness of p.o. reduction.

Specialization of $pre(\alpha)$ for different Modles of Computation

- $pre(\alpha)$ includes transitions of processes that contain α and that can change some pc_i to a value from which α can execute
- if enabling condition for α involves shared variables, then $pre(\alpha)$ contains all other transitions that can change these shared variables
- if α involves sending/receiving to/from quere d, then all transitions from other processes sending or receiving to/from d are in $pre(\alpha$

Statically computing D for Different Model of Computation

D includes:

- pairs of transitions that share a variable that is changed by at least one of them
- pairs of transitions belonging to the same process i
  - transitions in P_i share "variable" pc_i
  - includes in particular pairs of transitions in $current_i(s)$ for given s and P_i
  - rendez-vous communication: common transition of two processes, all transitions of both processes included in D

not included in D

- pair of send and receive transitions in different processes referring to the same channel d. they can enable, but never diable each other

Ample Set Construction:

- candidate for $ample(s) : T_i(s)$
  - all transitions in $T_i(s)$ are interdependent
  - $ample(s)$ mus contain either all of the transitions in $T_i(S)$, or none of them
- select P_i such that $T_i(s) \neq \varnothing$ and use $T_i(s)$ as a candidate for $ample(s)$
- what if this candiate $ample(s)$ does not satisfy $C1$?
  - some transitions independent of $T_i(s)$ are executed, eventually enabling transition α that is dependent on $T_i(s)$
  - the interdependent transitions cannot be in T_i, since all the transitions of P_i are interdependent
- cases in which $ample(s)$ does not satisfy $C1$
  1. α belongs to some other process P_k
     - then necessarily $dep(T_i(s))$ includes a transition of some process P_k
     - this condition can easily be checked by examining D
  2. α belongs to P_i
     - suppose that some $\alpha \in T_i$, violating $C1$, is executed from state s'
     - transition on path $s, \ldots, s'$ are independent of $T_i(s)$ and hence, belong to other processes
     - hence $pc_i(s) = pc_i(s')$ and $\alpha \in current_i(s)$
     - also, $\alpha \notin T_i(s)$ otherwise no $C1$ violation
     - hence $\alpha incurrent_i(s) - T_i(s)$
     - $\alpha \notin T_i(s) \Rightarrow \alpha$ is diabled in s
     - therefore, transition in $pre(\alpha)$ must be in $s, \ldots s'$
     - then necessarily, $pre(current_i(s) - T_i(s))$ includes transition of processes other than P_i
     - can be checke efficiently
- if one of these tests fails:
  - discard $T_i(s)$ as candidate
  - try other process, i.e. $T_k(s)$
- approach is over-approximating:
  ample sets might be discarded even though they would not acutally violate $C1$ if the program was executed dynamically