Introduction to MCRL2
Process modelling and verification

Luís Soares Barbosa

Universidade do Minho

HASLab
HIGH-ASSURANCE SOFTWARE LABORATORY

INL
INTERNATIONAL IBERIAN NANOTECHNOLOGY LABORATORY

UNITED NATIONS UNIVERSITY
UNU-EGOV

Universidade do Minho

mCRL2: A toolset for process algebra

mCRL2 provides:

- a generic process algebra, based on ACP (Bergstra & Klop, 82), in which other calculi can be embedded
- extended with data and (real) time
- with an axiomatic semantics
- the full μ-calculus as a specification logic
- powerful toolset for simulation and verification of reactive systems

www.mcrl2.org

Actions

Interaction through multisets of actions

- A multiaction is an elementary unit of interaction that can execute itself atomically in time (no duration), after which it terminates successfully

$$\alpha \quad ::= \quad \tau \mid a \mid a(d) \mid (\alpha \mid \alpha)$$

for $a \in N$.

- actions may be parametric on data
- the structure $\langle N, |, \tau \rangle$ forms an Abelian monoid

Sequential processes

Sequential, non deterministic behaviour

The set $\mathbb{P}$ of processes is the set of all terms generated by the following BNF, for α an action over N,

$$p ::= \alpha \mid \delta \mid p + p \mid p \cdot p \mid \mathrm{P}(d)$$

- atomic process: a for all $a \in N$
- choice: $+$
- sequential composition: $\cdot$
- inaction or deadlock: δ
- process references introduced through definitions of the form $\mathrm{P}(x : D) = p$, parametric on data

mCRL2: A toolset for process algebra

Example

```
act    order, receive, keep, refund, return;

proc   Buy = order.OrderedItem

       OrderedItem = receive.ReceivedItem + refund.Buy;
       ReceivedItem = return.OrderedItem + keep;

init   Buy;
```

Example

Clock

```
act    set, alarm, reset;

proc   P = set.R
       R = reset.P + alarm.R

init   P
```

Example

A refined clock

```
act    set:N, alarm, reset, tick;

proc   P = (sum n:N . set(n).R(n)) + tick.P
       R(n:N) = reset.P + ((n == 0) -> alarm.R(0) <> tick.R(n-1))

init   P
```

Parallel composition

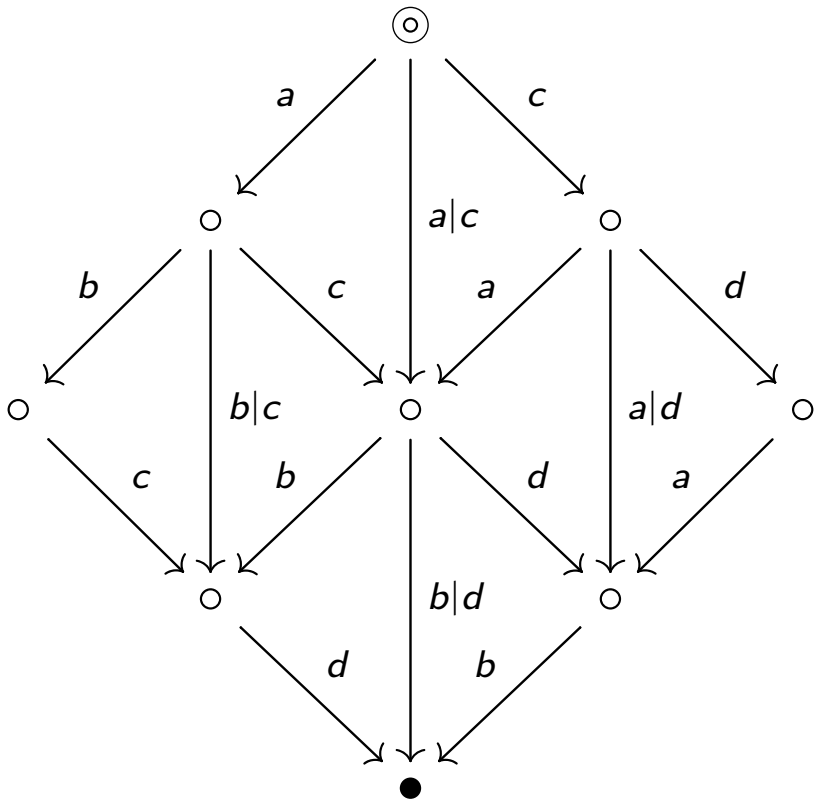
$\|$ = interleaving + synchronization

- modelling principle: interaction is the key element in software design
- modelling principle: (distributed, reactive) architectures are configurations of communicating black boxes
- MCRL2: supports flexible synchronization discipline ($\neq$ CCS)

$$p ::= \cdots \mid p \parallel p \mid p \mid p \mid p \between p$$

Parallel composition

An example



$$a \cdot b \parallel c \cdot d$$

Parallel composition

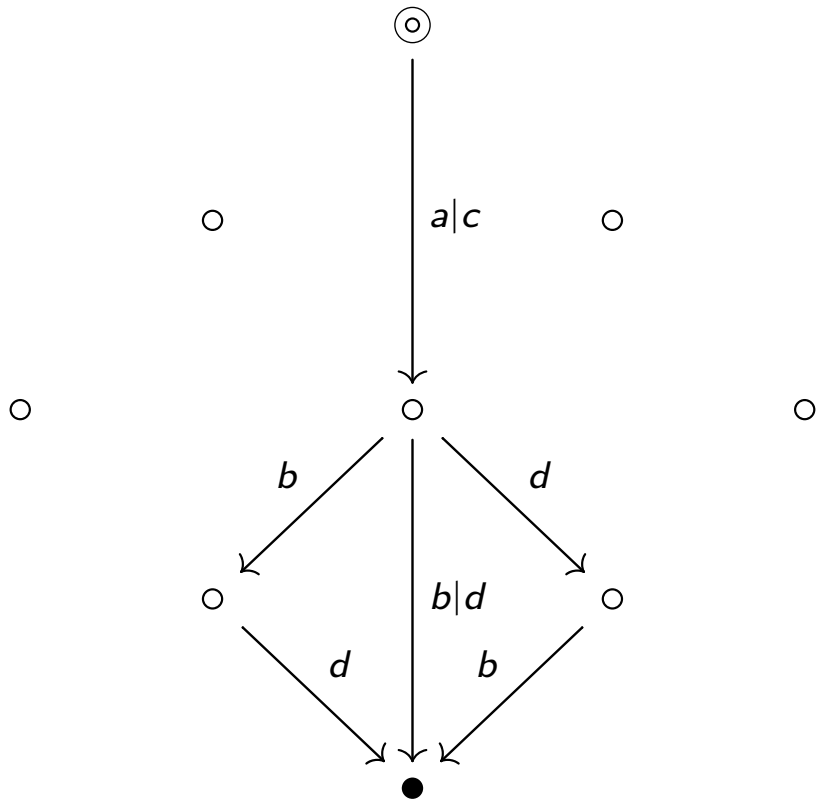
- parallel $p \parallel q$: interleaves and synchronises the actions of both processes.
- synchronisation $p \mid q$: synchronises the first actions of p and q and combines the remainder of p with q with $\parallel$, cf axiom:

$$(a.p) \mid (b.q) \sim (a \mid b).(p \parallel q)$$

- left merge $p \mathbin{\lfloor\!\lfloor} q$: executes a first action of p and thereafter combines the remainder of p with q with $\parallel$.

Parallel composition

An example



$$a \cdot b \mid c \cdot d$$

Interaction

Communication $\Gamma_C(p)$ (com)

- applies a communication function C forcing action synchronization and renaming to a new action:

$$a_1 \mid \cdots \mid a_n \rightarrow c$$

- data parameters are retained in action c, e.g.

$$\Gamma_{\{a|b\rightarrow c\}}(a(8) \mid b(8)) = c(8)$$
$$\Gamma_{\{a|b\rightarrow c\}}(a(12) \mid b(8)) = a(12) \mid b(8)$$
$$\Gamma_{\{a|b\rightarrow c\}}(a(8) \mid a(12) \mid b(8)) = a(12) \mid c(8)$$

- left hand-sides in C must be disjoint: e.g., $\{a \mid b \rightarrow c, a \mid d \rightarrow j\}$ is not allowed

Interface control

Restriction: $\nabla_B(p)$ (allow)

- specifies which actions are allowed to occur
- disregards the data parameters of actions

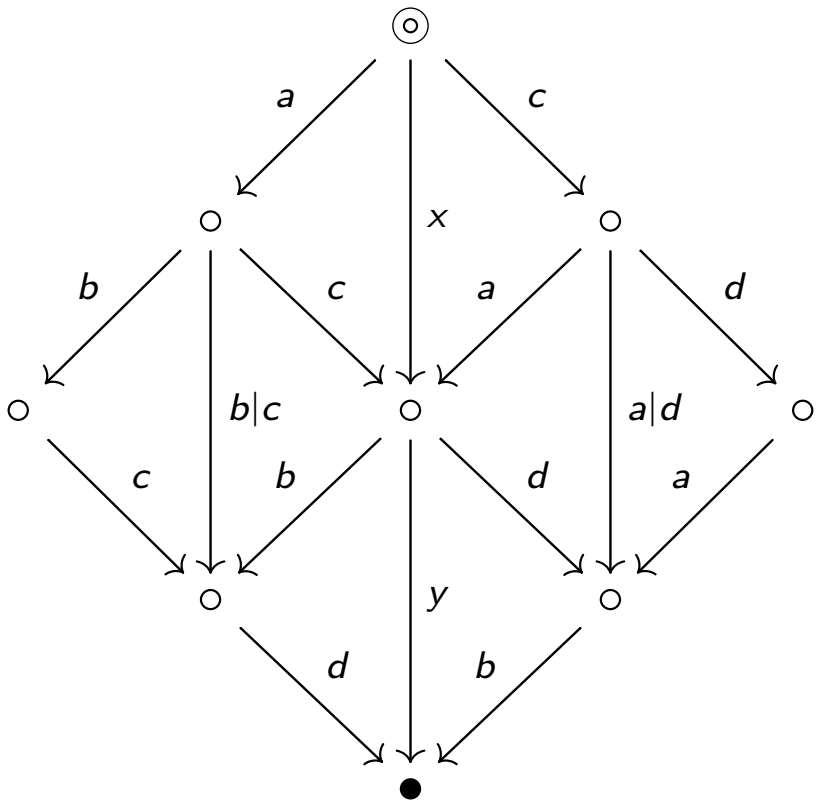
$$\nabla_{\{d,b|c\}}(d(12) + a(8) + (b(false, 4) \mid c)) \;=\; d(12) + (b(false, 4) \mid c)$$

- τ is always allowed to occur

Discuss: $\nabla_{\{x,y\}}(\Gamma_{\{a|c->x,b|d->y\}}(a.b \parallel c.d))$

Interface control

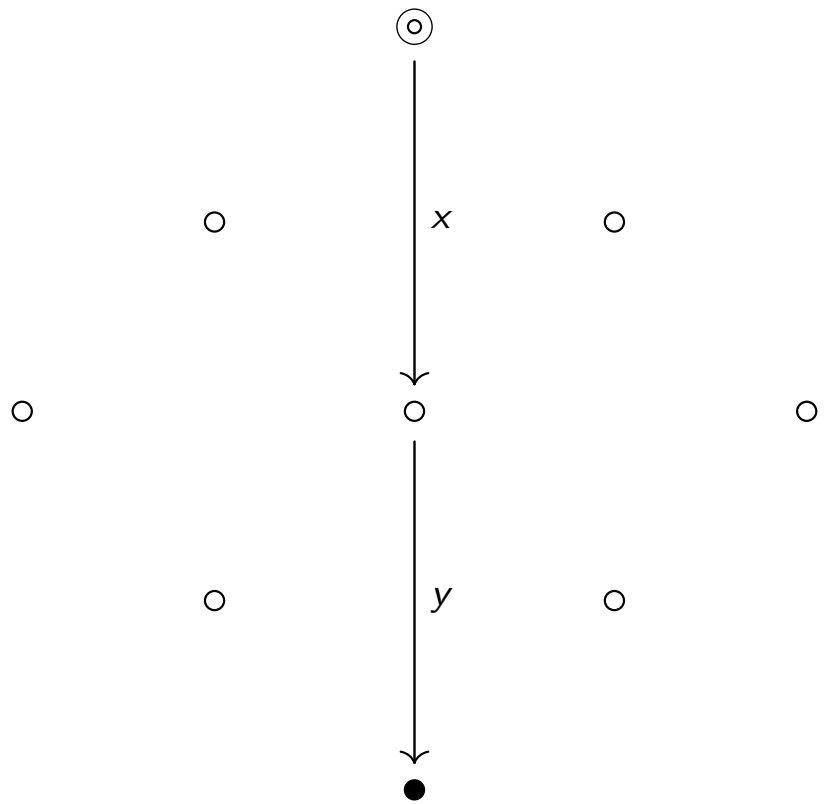
An example



$$\Gamma_{\{a|c->x,b|d->y\}}(a.b \parallel c.d)$$

Interface control

An example



$$\nabla_{\{x,y\}}(\Gamma_{\{a|c->x,b|d->y\}}(a.b \parallel c.d))$$

Interface control

Block: $\partial_B(p)$ (`block`)

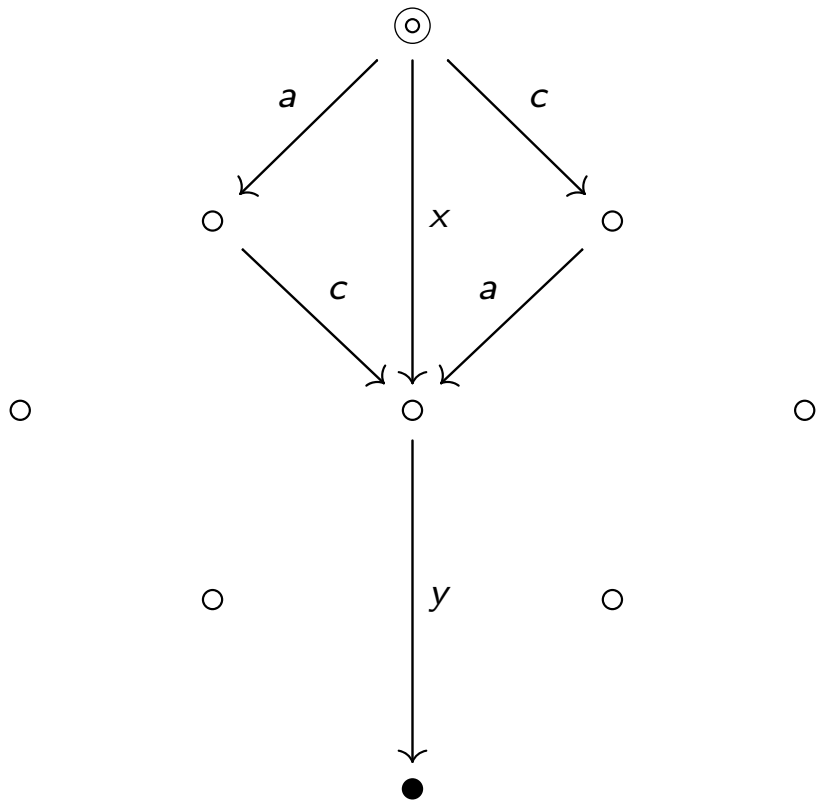
- specifies which actions are not allowed to occur
- disregards the data parameters of actions

$$\partial_{\{b\}}(d(12) + a(8) + (b(\mathit{false}, 4) \mid c)) \;=\; d(12) + a(8)$$

- the effect is that of renaming to δ
- τ cannot be blocked

Interface control

An example



$$\partial_{\{b,d\}}(\Gamma_{\{b|d->y\}}(a.b \parallel c.d))$$

Interface control

Enforce communication

- $\nabla_{\{c\}}(\Gamma_{\{a|b\rightarrow c\}}(p))$
- $\partial_{\{a,b\}}(\Gamma_{\{a|b\rightarrow c\}}(p))$

Interface control

Renaming $\rho_M(p)$ (`rename`)

- renames actions in p according to a mapping M
- also disregards the data parameters, but when a renaming is applied the values of data parameters are retained:

$$\begin{aligned}\rho_{\{d\to h\}}(d(12)+s(8)\mid d(\mathit{false})+d.a.d(7)) \\ = \; h(12)+s(8)\mid h(\mathit{false})+h.a.h(7)\end{aligned}$$

- τ cannot be renamed

Interface control

Hiding $\tau_H(p)$ (`hide`)

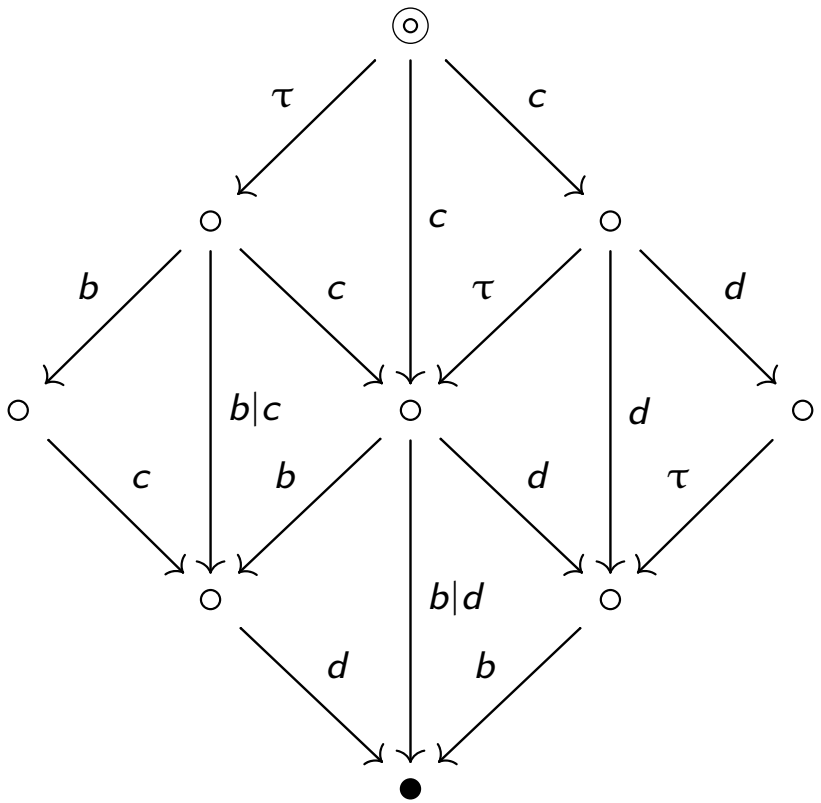
- hides (or renames to τ) all actions in H in all multiactions of p.
- disregards the data parameters

$$\begin{aligned}&\tau_{\{d\}}(d(12)+s(8)\mid d(\mathit{false})+h.a.d(7))\\ &\quad=\ \tau+s(8)\mid\tau+h.a.\tau\ =\ \tau+s(8)+h.a.\tau\end{aligned}$$

- τ and δ cannot be renamed

Interface control

An example



$$\tau_{\{a\}}(\Gamma_{\{b|d->y\}}(a.b \parallel c.d))$$

Example

New buffers from old

```
act    inn,outt,ia,ib,oa,ob,c :  Bool;

proc   BufferS = sum n:  Bool.inn(n).outt(n).BufferS;

       BufferA = rename({inn -> ia, outt -> oa}, BufferS);
       BufferB = rename({inn -> ib, outt -> ob}, BufferS);

       S = allow({ia,ob,c}, comm({oa|ib -> c}, BufferA || BufferB));

init   hide({c}, S);
```

Data types

- Equalities: equality, inequality, conditional (`if(-,-,-)`)
- Basic types: booleans, naturals, reals, integers, ... with the usual operators
- Sets, multisets, sequences ... with the usual operators
- Function definition, including the λ-notation
- Inductive types: as in

```
sort   BTree = struct leaf(Pos) | node(BTree, BTree)
```

Signatures and definitions

Sorts, functions, constants, variables ...

```
sort   S, A;

cons   s,t:S, b:set(A);

map    f:  S x S -> A;
       c:  A;

var    x:S;

eqn    f(x,s) = s;
```

Signatures and definitions

A full functional language ...

```
sort   BTree = struct leaf(Pos) | node(BTree, BTree);

map    flatten:  BTree -> List(Pos);

var    n:Pos, t,r:BTree;

eqn    flatten(leaf(n)) = [n];
       flatten(node(t,r)) = flatten(t) ++ flatten(r);
```

Processes with data

Why?

- Precise modeling of real-life systems
- Data allows for finite specifications of infinite systems

How?

- data and processes parametrized
- summation over data types: $\sum_{n:N} s(n)$
- processes conditional on data: $b \rightarrow p \diamond q$

Examples

A counter

```
act     up, down;
        setcounter:Pos;

proc    Ctr(x:Pos) = up.Ctr(x+1)
                   + (x>0) -> down.Ctr(x-1)
                   + sum m:Pos.(setcounter(m).Ctr(m))

init    Ctr(345);
```

Examples

A dynamic binary tree

```
act    left,right;

map    N:Pos;

eqn    N = 512;

proc   X(n:Pos)=(n<=N)->(left.X(2*n)+right.X(2*n+1))<>delta;

init   X(1);
```

Verification

System's correctness wrt a specification

- equivalence checking (between two designs), through $\sim$ and $=$
- unsuitable to check properties such as

  can the system perform action α followed by β?

  which are best answered by exploring the process state space

Verification

The verification problem

- Given a specification of the system's behaviour is in MCRL2
- and the system's requirements are specified as properties in a temporal logic,
- a model checking algorithm decides whether the property holds for the model: the property can be verified or refuted;
- sometimes, witnesses or counter examples can be provided

Which logic?

μ-calculus with data, time and regular expressions

Hennessy-Milner logic

... propositional logic with action modalities

Syntax

$$\phi \ ::= \ \mathit{true} \mid \mathit{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle K \rangle \phi \mid [K]\phi$$

Semantics: $E \models \phi$

$$
\begin{array}{lll}
E \models \mathit{true} & & \\
E \not\models \mathit{false} & & \\
E \models \phi_1 \wedge \phi_2 & \text{iff} & E \models \phi_1 \ \wedge \ E \models \phi_2 \\
E \models \phi_1 \vee \phi_2 & \text{iff} & E \models \phi_1 \ \vee \ E \models \phi_2 \\
E \models \langle K \rangle \phi & \text{iff} & \exists_{F \in \{E' \mid E \xrightarrow{a} E' \wedge a \in K\}} \cdot F \models \phi \\
E \models [K]\phi & \text{iff} & \forall_{F \in \{E' \mid E \xrightarrow{a} E' \wedge a \in K\}} \cdot F \models \phi
\end{array}
$$

Examples

Process logic: The taxi network example

- $\phi_0 =$ *In a taxi network, a car can collect a passenger or be allocated by the Central to a pending service*
- $\phi_1 =$ *This applies only to cars already on service*
- $\phi_2 =$ *If a car is allocated to a service, it must first collect the passenger and then plan the route*
- $\phi_3 =$ *On detecting an emergence the taxi becomes inactive*
- $\phi_4 =$ *A car on service is not inactive*

Examples

Process logic: The taxi network example

- $\phi_0 = \langle rec, alo \rangle true$
- $\phi_1 = [onservice]\langle rec, alo \rangle true$ or
  $\phi_1 = [onservice]\phi_0$
- $\phi_2 = [alo]\langle rec \rangle \langle plan \rangle true$
- $\phi_3 = [sos][-]false$
- $\phi_4 = [onservice]\langle - \rangle true$

Process logic: typical properties

- inevitability of a: $\langle - \rangle \mathit{true} \wedge [-a] \mathit{false}$
- progress: $\langle - \rangle \mathit{true}$
- deadlock or termination: $[-] \mathit{false}$
- what about

  $$\langle - \rangle \mathit{false} \quad \text{and} \quad [-] \mathit{true} \quad ?$$

- satisfaction decided by unfolding the definition of $\models$: no need to compute the transition graph

Example

$$\begin{aligned} Sem &\widehat{=} \; get.put.Sem \\ P_i &\widehat{=} \; \overline{get}.c_i.\overline{put}.P_i \\ S &\widehat{=} \; (Sem \mid (|_{i \in I} \; P_i)) \backslash_{\{get,put\}} \end{aligned}$$

- $Sem \models \langle get \rangle true$ holds because

$$\exists_{F \in \{Sem' \mid Sem \xrightarrow{get} Sem'\}} . \; F \models true$$

  with $F = put.Sem$.

- However, $Sem \models [put] false$ also holds, because
  $T = \{Sem' \mid Sem \xrightarrow{put} Sem'\} = \emptyset$.
  Hence $\forall_{F \in T} . \; F \models false$ becomes trivially true.

- The only action initially permmited to S is τ: $\models [-\tau] false$.

Example

$$\begin{aligned} Sem &\widehat{=} \; get.put.Sem \\ P_i &\widehat{=} \; \overline{get}.c_i.\overline{put}.P_i \\ S &\widehat{=} \; (Sem \mid (|_{i \in I} \; P_i))\backslash_{\{get,put\}} \end{aligned}$$

- Afterwards, S can engage in any of the critical events $c_1, c_2, ..., c_i$: $[\tau]\langle c_1, c_2, ..., c_i \rangle true$
- After the semaphore initial synchronization and the occurrence of c_j in P_j, a new synchronization becomes inevitable: $S \models [\tau][c_j](\langle - \rangle true \wedge [-\tau] false)$

Exercise

Verify:

$$\neg\langle a\rangle\phi = [a]\neg\phi$$
$$\neg[a]\phi = \langle a\rangle\neg\phi$$
$$\langle a\rangle \mathit{false} = \mathit{false}$$
$$[a]\mathit{true} = \mathit{true}$$
$$\langle a\rangle(\phi \vee \psi) = \langle a\rangle\phi \vee \langle a\rangle\psi$$
$$[a](\phi \wedge \psi) = [a]\phi \wedge [a]\psi$$
$$\langle a\rangle\phi \wedge [a]\psi \Rightarrow \langle a\rangle(\phi \wedge \psi)$$

Modal Equivalence

For each (finite or infinite) set Γ of formulae,

$$E \equiv_\Gamma F \quad \Leftrightarrow \quad \forall_{\phi \in \Gamma} \,.\, E \models \phi \Leftrightarrow F \models \phi$$

Examples

$$a.b.\,0 + a.c.0 \;\equiv_\Gamma\; a.(b.\,0 + c.0)$$

for $\Gamma \;=\; \{\langle x_1\rangle\langle x_2\rangle \ldots \langle x_n\rangle \mathit{true} \mid x_i \in \mathit{Act}\}$

(what about $\equiv_\Gamma$ for $\Gamma = \{\langle x_1\rangle\langle x_2\rangle\langle x_3\rangle \ldots \langle x_n\rangle[-]\mathit{false} \mid x_i \in \mathit{Act}\}$?)

Modal Equivalence

For each (finite or infinite) set Γ of formulae,

$$E \equiv_\Gamma F \quad \Leftrightarrow \quad \forall_{\phi \in \Gamma} \,.\, E \models \phi \Leftrightarrow F \models \phi$$

Examples

$$a.b.\,0 + a.c.0 \;\equiv_\Gamma\; a.(b.\,0 + c.0)$$

for $\Gamma \;=\; \{\langle x_1\rangle\langle x_2\rangle \ldots \langle x_n\rangle \mathit{true} \mid x_i \in \mathit{Act}\}$

(what about $\equiv_\Gamma$ for $\Gamma = \{\langle x_1\rangle\langle x_2\rangle\langle x_3\rangle \ldots \langle x_n\rangle[-]\mathit{false} \mid x_i \in \mathit{Act}\}$?)

Modal Equivalence

For each (finite or infinite) set Γ of formulae,

$$E \equiv F \quad \Leftrightarrow \quad E \equiv_\Gamma F \text{for every set } \Gamma \text{ of well-formed formulae}$$

Lemma

$$E \sim F \quad \Rightarrow \quad E \equiv F$$

Note

the converse of this lemma does not hold, e.g. let

- $A \widehat{=} \sum_{i \geq 0} A_i$, where $A_0 \widehat{=} 0$ and $A_{i+1} \widehat{=} a.A_i$
- $A' \widehat{=} A + \underline{fix}\ (X = a.X)$

$$\neg(A \sim A') \quad \text{but} \quad A \equiv A'$$

Modal Equivalence

Theorem [Hennessy-Milner, 1985]

$$E \sim F \quad \Leftrightarrow \quad E \equiv F$$

for image-finite processes.

Image-finite processes

E is image-finite iff $\{F \mid E \xrightarrow{a} F\}$ is finite for every action $a \in Act$

Modal Equivalence

Theorem [Hennessy-Milner, 1985]

$$E \sim F \quad \Leftrightarrow \quad E \equiv F$$

for image-finite processes.

Image-finite processes

E is image-finite iff $\{F \mid E \xrightarrow{a} F\}$ is finite for every action $a \in Act$

Modal Equivalence

Theorem [Hennessy-Milner, 1985]

$$E \sim F \quad \Leftrightarrow \quad E \equiv F$$

for image-finite processes.

proof

$\Rightarrow$: by induction of the formula structure

$\Leftarrow$: show that $\equiv$ is itself a bisimulation, by contradiction

Is Hennessy-Milner logic expressive enough?

Is Hennessy-Milner logic expressive enough?

- It cannot detect deadlock in an arbitrary process
- or general safety: all reachable states verify ϕ
- or general liveness: there is a reachable states which verifies ϕ
- ...

... essentially because

formulas in cannot see deeper than their modal depth

Is Hennessy-Milner logic expressive enough?

Example

$$\phi \;=\; \text{a taxi eventually returns to its Central}$$

$$\phi \;=\; \langle reg\rangle true \vee \langle -\rangle\langle reg\rangle true \vee \langle -\rangle\langle -\rangle\langle reg\rangle true \vee \langle -\rangle\langle -\rangle\langle -\rangle\langle reg\rangle true \vee \ldots$$

Revisiting Hennessy-Milner logic

Adding regular expressions

ie, with regular expressions within modalities

$$\rho ::= \epsilon \mid \alpha \mid \rho.\rho \mid \rho + \rho \mid \rho^* \mid \rho^+$$

where

- α is an action formula and ϵ is the empty word
- concatenation $\rho.\rho$, choice $\rho + \rho$ and closures ρ^* and ρ^+

Laws

$$\begin{aligned}
\langle \rho_1 + \rho_2 \rangle \phi &= \langle \rho_1 \rangle \phi \vee \langle \rho_2 \rangle \phi \\
[\rho_1 + \rho_2] \phi &= [\rho_1] \phi \wedge [\rho_2] \phi \\
\langle \rho_1 . \rho_2 \rangle \phi &= \langle \rho_1 \rangle \langle \rho_2 \rangle \phi \\
[\rho_1 . \rho_2] \phi &= [\rho_1][\rho_2] \phi
\end{aligned}$$

Revisiting Hennessy-Milner logic

Examples of properties

- $\langle \epsilon \rangle \phi \ = \ [\epsilon] \phi \ = \ \phi$
- $\langle a.a.b \rangle \phi \ = \ \langle a \rangle \langle a \rangle \langle b \rangle \phi$
- $\langle a.b + g.d \rangle \phi$

Safety

- $[-^*] \phi$
- it is impossible to do two consecutive enter actions without a leave action in between:
  $[-^*.enter.-leave^*.enter]false$
- absence of deadlock:
  $[-^*]\langle - \rangle true$

Revisiting Hennessy-Milner logic

Examples of properties

Liveness

- $\langle -^* \rangle \phi$
- after sending a message, it can eventually be received:
  $[send]\langle -^* . receive \rangle true$
- after a send a receive is possible as long as an exception does not happen:
  $[send . - excp^*]\langle -^* . receive \rangle true$

The real answer: The modal μ-calculus

Intuition

- look at modal formulas as set-theoretic combinators
- introduce mechanisms to specify their fixed points
- introduced as a generalisation of Hennessy-Milner logic for processes to capture enduring properties.

References

- Original reference: *Results on the propositional μ-calculus*, D. Kozen, 1983.
- Introductory text: *Modal and temporal logics for processes*, C. Stirling, 1996

The modal μ-calculus

The modal μ-calculus

- modalities with regular expressions are not enough in general
- ... but correspond to a subset of the modal μ-calculus [Kozen83]

Add explicit minimal/maximal fixed point operators to Hennessy- Milner logic

$$\phi ::= X \mid \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi\wedge\phi \mid \phi\vee\phi \mid \phi\Rightarrow\phi \mid \langle a\rangle\phi \mid [a]\phi \mid \mu X\,.\,\phi \mid \nu X\,.\,\phi$$

The modal μ-calculus

The modal μ-calculus (intuition)

- $\mu X \,.\, \phi$ is valid for all those states in the smallest set X that satisfies the equation $X = \phi$ (finite paths, liveness)
- $\nu X \,.\, \phi$ is valid for the states in the largest set X that satisfies the equation $X = \phi$ (infinite paths, safety)

Warning

In order to be sure that a fixed point exists, X must occur positively in the formula, ie preceded by an even number of negations.

The modal μ-calculus

Laws & Notes

$$\mu X \,.\, \phi \;\Rightarrow\; \nu X \,.\, \phi$$

and self-duals:

$$\begin{aligned} \neg \mu X \,.\, \phi &= \nu X \,.\, \neg \phi \\ \neg \nu X \,.\, \phi &= \mu X \,.\, \neg \phi \end{aligned}$$

Translation of regular formulas with closure

$$\begin{aligned} \langle R^* \rangle \phi &= \mu X \,.\, \langle R \rangle X \vee \phi \\ [R^*] \phi &= \nu X \,.\, [R] X \wedge \phi \\ \langle R^+ \rangle \phi &= \langle R \rangle \langle R^* \rangle \phi \\ [R^+] \phi &= [R][R^*] \phi \end{aligned}$$

Example: The dining philosophers problem

Formulas to verify

- No deadlock (every philosopher holds a left fork and waits for a right fork (or vice versa):

```
[true*]<true>true
```

- No starvation (a philosopher cannot acquire 2 forks):

```
forall p:Phil.  [true*.!eat(p)*] <!eat(p)*.eat(p)>true
```

- A philosopher can only eat for a finite consecutive amount of time:

```
forall p:Phil.  nu X. mu Y. [eat(p)]Y && [!eat(p)]X
```

- there is no starvation: for all reachable states it should be possible to eventually perform an `eat(p)` for each possible value of `p:Phil`.

```
[true*](forall p:Phil.  mu Y. ([!eat(p)]Y && <true>true))
```