ExSpect course

Contents:

- Introduction
- Static Part
- Dynamic Part
- Modularity
- Examples

Introduction

Constructive model:
system composed of interfacing subsystems
Example: production control.

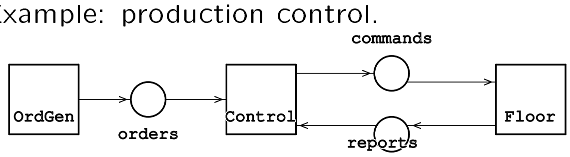

fig.1

Orders are sent by (`OrdGen` to `Control`. `Control` sends commands to the work floor (`Floor`) and receives reports back.

Channels are typed: `order` has type `ordtype`.

We "open" the `Control` subsystem. Subsystems have "pins" for connection. We also see a data store.

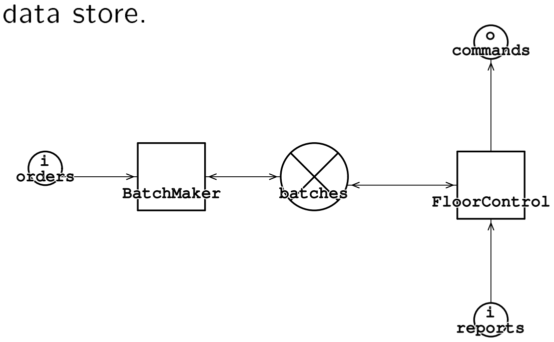

fig.2

The system is divided into a part (`BatchMaker`) that receives orders and a part (`FloorControl`) that communicates with the workfloor.
Their interface is a store (`batches`) that contain batches waiting to be processed.

We open the `BatchMaker`. The store is connected to a "store pin". An internal store and an elementary subsystem ("processor") appear.

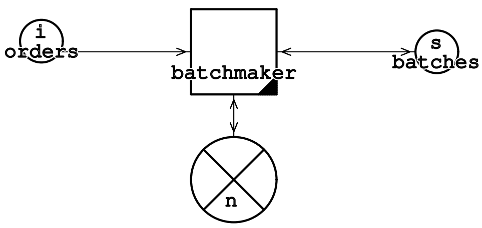

fig.3

Processors have a behavior that is functional.

ExSpect processor batchmaker similar to Z.
Differences:
"assignments" instead of equality.
If-then-else-fi construction.
No assignment for unaltered store.

in:	`orders`	`ordtype`
out:		
store:	`batches`	`$btype`
	`n`	`num`

Body:

```
if addset(batches,orders) != {}
then batches <- bupd (
 (batches,
  first(addset(batches,orders)),
  orders)
else batches <-
  badd (batches, orders, n),
  n <- n+1
fi
```


ExSpect possesses *static* and *dynamic* component.

Static component describes math. objects (set, function, relation)
terms: `ordtype`,
`addset(batches,orders) != {}`

Dynamic component describes world objects in terms of math. objects
processors, channels, stores, (sub)systems

Process model: Petri nets!

Static part

ExSpect static:
typed λ calculus: terms with types.

type system:
- basic types (sets of basic terms)
- type constructors (fu. from sets to sets)

together: type constructions
used for: type definitions
set of type constructions: *TYPE*

σ: maps type construction onto set of terms. The terms within $\sigma(T)$ for any type construction T are called *values*.

Basic types: `void`, `bool`, `str`, `num`, `real`

$$\begin{aligned}
\sigma(\texttt{void}) &= \{\} \\
\sigma(\texttt{bool}) &= \{\texttt{true}, \texttt{false}\} \\
\sigma(\texttt{str}) &= \{\texttt{''}, \texttt{'!'}, \texttt{''''}, \ldots, \\
&\quad \texttt{'ExSpect'}, \ldots\} \\
\sigma(\texttt{num}) &= \{0, 1, \texttt{-}1, \ldots, \texttt{-}322/5, \ldots\} \\
\sigma(\texttt{real}) &= \{0.0, \ldots, \texttt{-}55.19452, \ldots\}
\end{aligned}$$

Types `real` and `num` are both numeric; values of `real` are *approximations*.

$$2 * 2 = 4$$
$$2.0 * 2.0 \approx 4.0$$

Type constructors ($A, B, \ldots$ are types):
$\underline{\$}A$, $\underline{*}A$: set/list constructor,
$A\underline{><}B$: pair constructor,
$A\underline{->}B$: map constructor,
$\underline{[}l_1\underline{:}A_1 \ldots l_n\underline{:}A_n\underline{]}$: record constructor.

σ for constructed types: ($n \geq 0$

$$\begin{array}{l}
\sigma(\underline{\$}G) = \{\underline{\{}g_1 \ldots g_n\underline{\}} \mid g_i \in \sigma(G)\} \\
\sigma(\underline{*}G) = \{\underline{<|}g_1 \ldots g_n\underline{|>} \mid g_i \in \sigma(G)\} \\
\sigma(G\underline{><}H) = \{\underline{<<}g\underline{,}h\underline{>>} \mid g \in \sigma(G) \wedge h \in \sigma(H)\} \\
\sigma(G\underline{->}H) = \\
\quad \{\underline{\{<<}g_1\underline{,}h_1\underline{>>} \ldots \underline{<<}g_n\underline{,}h_n\underline{>>\}} \\
\quad\; \mid g_i \in \sigma(G) \wedge h_i \in \sigma(H) \wedge g_i = g_j \Rightarrow h_i = h_j\} \\
\sigma(\underline{[}l_1\underline{:}G_1 \ldots l_n\underline{:}G_n\underline{]} = \\
\quad \{\underline{[}l_1\underline{:}g_1 \ldots l_n\underline{:}g_n\underline{]} \mid g_i \in \sigma(G_i)\}
\end{array}$$

Note: $\sigma(G\texttt{->}H) \subseteq \sigma(\texttt{\$(}G\texttt{><}H\texttt{)})$.

A value v has type T iff $v \in \sigma(T)$.
A value v may have several types, but it has one strongest type $\tau(v)$.
One has: $v \in \sigma(T) \Rightarrow v \in \tau(v)$.
`{}` has type `$num`,`$str` etc. $\tau(\texttt{\{\}}) =$ `$void`.

Values $\{g_1, g_2, \ldots\}$ represent sets:
$\{S\ x, y\ S'\} = \{S\ y, x\ S'\}$
$\{S\ x, x\ S'\} = \{S\ x\ S'\}$

Record values of the form $[l_1{:}g_1, \ldots]$ are not order-dependent:
$[L\ l_1{:}g_1, l_2{:}g_2\ L'] = [L\ l_2{:}g_2, l_1{:}g_1\ L']$
They may not have duplicate labels ($l_1 \neq l_2$).
The same holds of course for record types.

type definitions are of the form
$\underline{\texttt{type}}$ *name* $\underline{\texttt{:=}}$ *type*

consequence: $\sigma(name) = \sigma(\textit{type})$

Example:

```
type coord := real><real;
type node := str;
type graph := node->(coord><$node);
-- directed graph,
-- location and descendents per node
```


Exercises:

1. Give correctness & strongest type.

`2.56.7`	wrong syntax
`{false,true}`	`$bool`
`<<1,{7}>>`	`num><$num`
`{<<1,{7}>>}`	`num->$num`
	`$(num><$num)` is weaker
`<|1,{7}|>`	illegal
`[a:1,b:{}]`	`[a:num,b:$void]`
`[b:<||>,x]`	wrong syntax
`{{}}`	`$$void`
`<<<<4,''>>,'1'>>`	`(num><str)><str`

2. Define type for chess tournament results.

3. Give number of elements of type `bool->bool`

2.

```
type comp := str
type chtres := $[wh:comp, bl:comp, res:num]
([wh:comp, bl:comp] -> num
comp -> $[had_wh:bool, opp:comp, res:num])
```

It depends on planned use which is best.

3.
9 elements: the empty mapping, 4 singletons, 4 doubletons.

Terms: values, variables, constructions.
Term constructors: (basic) functions, quantors.
Expressions (terms containing variables) are used to define new functions.
First: basic functions.

ExSpect functions transform list (0 or more) of input values into output value.

Example:
`4 elt {1,3,4}` $=$ `true`.
`elt`: function name,
`4`,`{1,3,4}`: input values (parameters),
`true`: output value (result)

Function f has *signature* $\tau(f)$
Signature: partial function of type-list to types.
If $\sigma(\tau(f)) = F$ and $f(e_1, \ldots, e_n) = e_0$ then $(\tau(e_1), \ldots, \tau(e_n)) \in \mathrm{dom}(F)$ and
$\tau(e_0) = F(\tau(e_1), \ldots, \tau(e_n))$.
This is checked by the type checker.

Simple functions: singleton signature.

$\tau(\mathtt{sin}) = \{\mathtt{real} \mapsto \mathtt{real}\}$
`sin(3.1459)` $\approx$ `-0.0043`

$\tau(\mathtt{and})) = \{(\mathtt{bool}, \mathtt{bool}) \mapsto \mathtt{bool}\}$
`true and false` $=$ `false`

Infix notation for binary functions allowed.

Overloaded functions
(finite signature of size > 1), e.g.
$\tau(\mathtt{sub})) =$
$\{(\mathtt{num}, \mathtt{num}) \mapsto \mathtt{num}, (\mathtt{real}, \mathtt{real}) \mapsto \mathtt{real}\}$
`sub(12/5,4/3)` $=$ `12/5 - 4/3` $=$ `16/15`
`sub(2.4,1.333)` $=$ `2.4 - 1.333` $\approx$ `1.067`

Many arithmetical functions in this category (but `div` and `mod` are simple).
Standard infix notation for arithmetical constructions, e.g. `5*(4-3)`.

Polymorphic functions: infinite signature , e.g.
$\tau(\texttt{union}) = \{(\$T, \$T) \mapsto \$T \mid T \in \mathit{TYPE}\}$
{1,2,3} union {3,4,5} = {1,2,3,4,5}
{} union {'3','4'} = {'3','4'}

(un)equality test (a = b, a != b),
conditional fu. (if a then b else c fi),
selection from set (pick(a), rest(a):
deterministic, underspecified!),
pair projections (pi1, pi2),
domain/range of mappings (dom,rng).

Polymorphy and overloading combined, e.g.
$\tau(\texttt{head}) =$
$\{*T \mapsto T \mid T \in \mathit{TYPE}\} \cup \{\texttt{str} \mapsto \texttt{str}\}$
head(<|1,2,3|>) = 1, head('abcd') = 'a'.
Also: apply, ins, tail, cat. Abbreviation M.a
for apply(M,a).

Special polymorphic functions, e.g. set extension.
$\tau(\{\ldots\}) = \{T \mapsto \$T \mid T \in \mathit{TYPE}\} \cup$
$\{(T, T) \mapsto \$T \mid T \in \mathit{TYPE}\} \cup \ldots$
{1,9-5,2*2} = {1,4}

Record extensions:
$\tau([l_1{:}\_, \ldots, l_n{:}\_]) =$
$\{(T_1, \ldots, T_n) \mapsto [l_1{:}T_1, \ldots, l_n{:}T_n] \mid$
$T_1 \in \mathit{TYPE} \ \wedge \ \ldots \ \wedge \ T_n \in \mathit{TYPE}\}$
[a:5*7, b:head('aap')] = [a:35,b:'a']

Record projections:
$\tau(\_@l) = \{[l{:}T, \ldots] \mapsto T \mid T \in \mathit{TYPE}\}$
[a:35,b:'a'] @ a = 35

"Errors" signalled by term abort.

runtime error	explanation
1/0	division by zero
head('')	head empty string
pick({})	pick on empty set
tail(<||>)	tail of empty list
{<<1,2>>,<<3,4>>}.2	invalid apply arg.

Propagation of abort:

<<x,abort>> = <<abort,y>> = abort
{...abort...} = abort
[... l:abort...] = abort

Exceptions: cond, and, or, impl
false and abort evaluates to false
(but abort and false aborts!)

Exercises:
1. Evaluate if possible.

if not(2=1) then 6-1 else 0	syntax error
<|1,4,1|> - 1	wrong type
tail('a') elt {tail('b')}	true
if 6=7 then 6/0 else 5.1 fi	wrong type
{0, 4+1, 5}	{0,5}
[a:head('a')]@b	wrong type
[a:'', b:1/(1-1)] upd [c:1.]	abort
5.41+0.99=6.4	true probably
pi1(<<8-4,1/0>>)	abort

2. Always true? If not, counterexample!
(a,b:num, s:str, A:$num).

(head(s) cat tail(s)) = s	s = ''
not(a elt {b})	a = b
not(a elt A) or ({a} union A = A)	OK
a/2+a/2=a and a*(1/a)=1	a = 0

Mapping intension: [x:A|E_x].
x: identifier
A: set term (of type \$$B$)
E_x: term with variable x of type B.
τ([x:A|E_x]) = B -> $\tau(E_x)$
Value: {<<a_1,E_{a_1}>>,...,<<a_n,E_{a_n}>>},
where A = {a_1,...,a_n} and E_a obtained by substitution.

Mapping intension combined with quantifiers.
(E any term, S set, N numeric, P predicate)

ExSpect	math
[x:A|E_x]	$\{(x, E_x) \mid x \in A\}$
rng[x:A|E_x]	$\{E_x \mid x \in A\}$
all[x:A|P_x]	$\forall x \in A : P_x$
any[x:A|P_x]	$\exists x \in A : P_x$
\$[x:A|P_x]	$\{x \mid x \in A \wedge P_x\}$
sum[x:A|N_x]	$\Sigma_{x \in A} N_x$
max[x:A|N_x]	$\text{MAX}_{x \in A} N_x$
min[x:A|N_x]	$\text{MIN}_{x \in A} N_x$
union[x:A|S_x]	$\bigcup_{x \in A} S_x$
rng[x:\$[x:A|P_x]|E_x]	$\{E_x \mid x \in A \wedge P_x\}$

Function definition: function name, parameter names and types, result type, body term.

```
half := 1/2: num
triangle[x:num] := x*(x+1)/2: num
mult[x:$num,y:num] :=
  rng[t:x|t*y]: $num
```

τ(half) = num
τ(mult) = {(\$num, num) $\mapsto$ \$num}
{5,7} mult 3 = {5,7}*3 = {15,21}

Last definition overloads multiplication (*) symbol.

Definition of polymorphic function by means of type variables (T,S,R):
```
isect[x:$T,y:$T] := $[t:x|t elt y]: $T
```

τ(isect) = {(\$$T$, \$$T$) $\mapsto$ \$$T$ | $T \in$ *TYPE*}
{1,2,3} isect {2,3,4} = {2,3}

Overloaded functions are created by including several definitions with the same name.
Type checker signals incorrect overloadings.

Example:

```
apply [x:T->S,y:T] :=
  pick($[t:x|pi1(t)=y]): S
apply [x:T->S,y:$T] :=
  rng[t:y| x.t]: $S
```

The term `apply([x:{{1,2},{3}}| x],{})` is ambiguous.

Type strengthening ("cast") may be needed.
```
upd[x:T->S,y:T->S] :=
 y union
  $[t:x| not(pi1(t) elt dom(y)]: T->S
upd[x:T->S,y:T,z:S] :=
 x upd {<<y,z>>}: T->S
```

For type checker, τ({<<y,z>>}) = \$(T><S) instead of T->S, thus: error.
The standard function `tomap` casts set-of-pairs types to corresponding mapping types. The defining term can be written as
`x upd tomap({<<y,z>>})`.

The cast `tovoid` bypasses the type checker.

Recursive definitions: used in body term.

```
size[x:$T] :=
  if x={} then 0
  else 1+size(rest(x)) fi: num
```

Note: `size(x)` can be written `#x`.

Recursion is dangerous (nontermination!)
Many recursive definitions can be replaced by quantor.

```
addrs[x:$[name:str,addr:str]] :=
  if x={} then {}
  else pick(x)@addr ins
       addrs(rest(x)) fi: $str
```

Better body term: `rng[t:x|t@addr]`

Difference types / set terms:
context within language
type may be infinite, set terms always finite.
no "enumeration" types.
`bool`: type, `{false,true}`: term.
So `rng[x:bool| not(x)]` is incorrect,
`rng[x:{false,true}| not(x)]` is correct.

Difference functions / mappings:
mappings are finite, have type,
functions have signature.
`[x:{false,true}|not(x)]` (mapping),

```
not[x:bool] :=
 if x then false else true fi: bool
```

(function).

Exercises:

1.Define a function `dommax[x:T->num]: $T`, such that `dommax(`M`)` contains for an appropriate M those domain elements for which M attains its maximum value. So e.g. `dommax[x:{1,2,3}|x*(x-4)]` = `{1,3}`.

2.Give recursive definition of `set` (= `$`) quantor.

Answers: (note that `[t:dom(x)|x.t]` = `x`)

```
dommax[x:T->num] :=
 $[t: dom(x)| x.t = max(x)]: $T
```

```
set[x:T->bool] :=
 if x={} then {}
 elif pi2(pick(x))
 then pi1(pick(x)) ins set(tomap(rest(x)))
 else set(tomap(rest(x))) fi: $T
```


Dynamic part

Processor definition consists of name, parameter names and types, precondition and body.

Parameters: input / output / store pins, other parameters.
Together: signature (polymorphy is possible)
Precondition: boolean term
Body: (conditional) statement list.

Body syntax:
csl ::= sl | <u>`if`</u> pred <u>`then`</u> csl <u>`else`</u> csl <u>`fi`</u>
pred ::= boolean term
sl ::= <u>`skip`</u> | ass | ass <u>`,`</u> sl
ass ::= id <u>`<-`</u> term <u>`delay`</u> real term

`else skip`, `delay 0.0` and precondition `true` can be omitted.
Concurrent assignments in statement list.

Examples:

```
name:  start
in:    a:  num, b:num
out:   c:num
val:   v:num
pre:   a>b-v and a<b+v
body:  c <- a+b

name:  finish
in:    c:num
out:   b:num, d:num
body:  if c<10 then b <- c
       else b <- c/2, d <- c/2 fi
```


Processors can be *installed* in a *system*:
input/output pins connected to channels
(multiple connections allowed)
store pins are connected to stores
other parameters filled in.

Type checking for correct installation:
input channels: subtype
output channels: supertype
stores/parameters: same type

Processors may be *polymorphic* by the use of a type variable (e.g. `T`). A polymorhic processor with input pin of `i` type `T` and output pin `o` of type `$T` can e.g. be installed with `i` connected to `num` channel and `o` connected to `$num` channel.

Example:
channels A,B,C of type `num`.
installation `start`: a-A, b-B, c-C, v-5

Installed processor is *automaton*:
If input channels receive objects matching the precondition, these objects are *consumed* and for output channels objects are *produced* matching the assignments. This is called *firing* of the processor.

Example:
Provide input: A-5,B-3
then output: C-8
Provide input: A-7,B-1
then no consumption and output.

A system can contain several processors sharing channels.

Example: system `small`

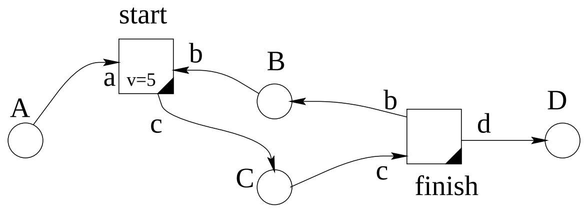

fig.4

Behaviour is given by graph with state (distribution of objects over channels) in nodes and firing of processor along edges. Each firing leads to a new state.

Behaviour for A-5,B-7 is diplayed.

$$A(5)B(7) \xrightarrow{\text{start}} C(12) \xrightarrow{\text{finish}} B(6)D(6)$$

fig.5

Different pins may be mapped on same channel.

Example: system `smrec`

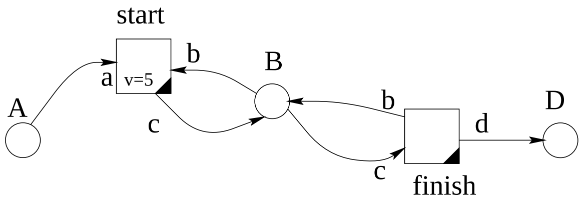

fig.6

Behaviour: note “choice” and “loop”.

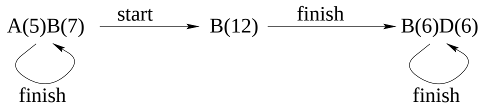

fig.7

System definition may have signature (like for processors). Such a system can be installed in higher-level system. Even polymorphy is possible.

Example: system `combi`
in `a,a1:  num`
out `d:  num`

init b: 0 (c: empty)
installation `start`: a-a, b-b, c-c, v-5
installation `start.2`: a-a1, b-b, c-c, v-7
installation `finish`: c-c, b-b, d-d

System `combi` is installed in `main`. Channels a,d of type `num`.
Subsystem `combi` is installed with a-a, a1-a, d-d.

Picture: `main-combi-start-finish`

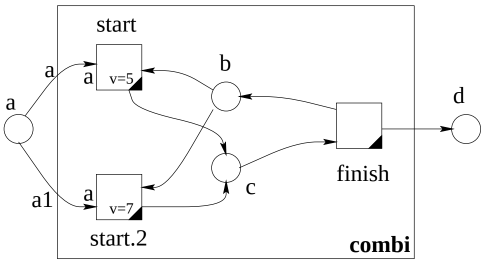

fig.8

Behaviour for a - 4,6

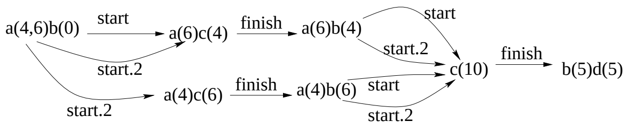

fig.9

Exercises:

1. Define a polymorphic processor with an input and output pin of type `T` and a store pin `s` of type `bool`. Whenever `s` has value `true`, the input is copied to the output with delay 1.0. When `s` has value `false`, the input is left untouched.

2. Define a subsystem with input pin of type `$T` and output pin of type `T` that converts an input set into its elements.

Answer1:

```
proc p[in i:T, out o:T, store s:bool | pre s]
 := o <- i delay 1.0
```

Answer2:

```
sys S[in i:$T, out o:T] :=
channel c:$T, copy(in i, out c),
iter(in c, out c, o)

proc iter [in c1:$T, out c2:$T, o:T] :=
if c1 != {}
then o <- pick(c1), c2 <- rest(c1) fi
```


Semantics of system without parameters: high-level Petri Net.

Petri Net is bipartite directed graph (P, T, I, O), with P, T sets (*places, transitions*),
$I, O : (T \times P) \rightarrow \mathbb{N}$ (i/o arc multiplicity).

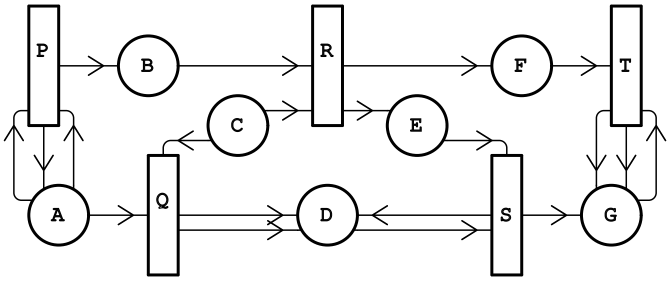

fig.10

$\mathcal{T}$: universe of tokens. A token has a value. Different tokens may have the same value.

A set of tokens in the net can be represented as an element of $S == P \rightarrow \mathbb{P}(\mathcal{T})$
For each transition $t : T$ a partial function $\rho_t : S \nrightarrow S$ relates consumed tokens to produced tokens, such that the number of consumed and produced tokens is determined by O, I respectively.
$\forall t \in T; p \in P; i \in \mathrm{dom}(\rho_t) :$
$\quad |i(p)| = I(t, p) \;\wedge\; |\rho_t(i)(p)| = O(t, p)$

Transition system relates $s, s' : S$ for $t : T$:
$s \xrightarrow{t} s' \Leftrightarrow \exists i \in \mathrm{dom}(\rho_t) : \forall p \in P : i(p) \subseteq s(p) \;\wedge$
$\quad s'(p) = (\rho_t(i)(p) \setminus i(p)) \cup s(p)$

Analysis on Petri nets:
number of tokens per place (projection of state) is vector, firing transition is linear transformation of it.

S-invariants: linear transformations of the above vector that remain invariant under each transition firing.
In Fig. 10 e.g. $|A| + |B| - |C|$ is invariant.
If P fires, 1 token less in A and 1 more in B.
If R fires, 1 token less in B and C.
If Q fires, 1 token less in A and C.

T-invariants: combination of firings giving equiv. state (same number of tokens per place).

S- and T-invariants can be easily computed by linear algebra. Net corresponds to matrix.

Net in Fig. 10 corresponds to following matrix.

	P	Q	R	S	T
A	−1	−1	0	0	0
B	1	0	−1	0	0
C	0	−1	−1	0	0
D	0	2	0	0	0
E	0	0	1	−1	0
F	0	0	1	0	−1
G	0	0	0	1	1

S-inv: lin. dependency rows
T-inv: lin. dependency columns

traffic light system

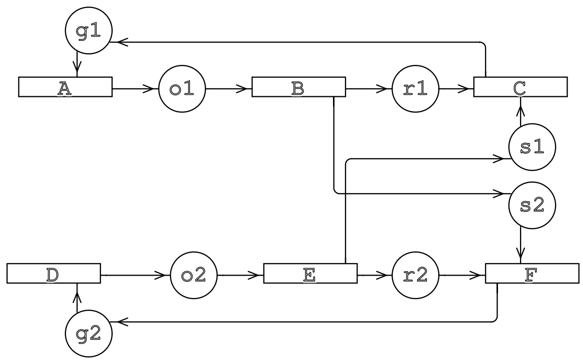

fig.11

Fig.11 depicts a traffic light system;
token in g_1: light crossing1 green etc.
Initial state: 1 token in g_1, 1 in r_2.

S-invariants
$g_1 + o_1 + r_1 = g_2 + o_2 + r_2 = 1$:
both crossings have 1 light burning
$r_1 + r_2 - s_1 - s_2 = 1$:
at least 1 light red

T-invariant: $A + B + C + D + E + F$.

ExSpect store: place with 1 token (S-invariant).
Exspect processor: group of transitions (number of output tokens may depend upon input).

Transformation ExSpect to Petri Net:

```
proc p [in i:num, out o:num,
        store s:num] :=
 if i>s then o<-i
 else s<-s-i fi

sys main :=
channel c:num init ...,
channel d:num init ...,
store s:num init ...,
p (in c, out d, store s)
```

This net becomes:

```
proc p_1 [in i:num, out o:num, s:num |
 pre i>s] := o<-i, s<-s

proc p_2 [in i:num, out s:num |
 pre not(i>s)] := s<-s-i

sys main :=
channel c:num init ...,
channel d:num init ...,
channel s:num init ...,
p_1 (in c,s, out d,s),
p_2 (in c,s, out s)
```


Extension to Petri nets: inhibitor arcs.
A transition connected by an inhibitor arc to a channel c may not fire if c is nonempty. Inhibitors are added at installation.

Semantics: extra component $H : T \to I\!P(P)$.
Extra requirement for firing:
$p \in H(t) \Rightarrow s(p) = \emptyset$

Possible uses: safe nets, priority.

Safe nets: places contain ≤ 1 token.

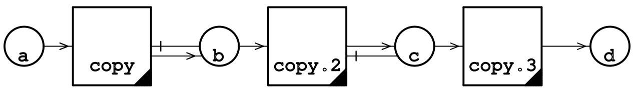

fig.12

Better: add extra place.

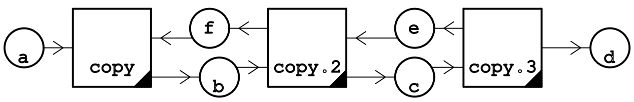

fig.13

Priority of shared resource a.

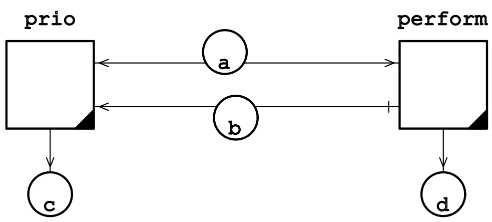

fig.14

Second extension: (simulated) time.
A token t has a timestamp $\theta(t)$,
system state contains simulation time Θ.
Only tokens t with $\theta(t) \leq \Theta$ can be consumed.
If (and only if) no transition can fire, Θ is incremented up to the first moment that some transition is enabled. Initially $\Theta = 0$.

Inhibition only by tokens with $\theta(t) \leq \Theta$.

Example:

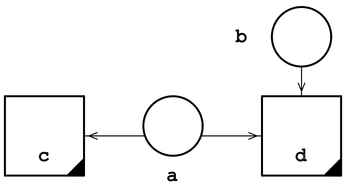

fig.15

Semantics:
State (s, Θ) from $(P \to \mathbb{P}(\mathcal{T})) \times \mathbb{R}$

$$(s,\Theta) \xrightarrow{t} (s',\Theta) \Leftrightarrow \exists i \in \mathrm{dom}(\rho_t) :$$
$$\forall p \in P : i(p) \subseteq \{\sigma \in s(p) \mid \theta(\sigma) \leq \Theta\} \wedge$$
$$p \in H(t) \Rightarrow \{\sigma \in s(p) \mid \theta(\sigma) \leq \Theta\} = \emptyset \wedge$$
$$s'(p) = (\rho_t(i)(p) \setminus i(p)) \cup s(p)$$

$$(s,\Theta) \xrightarrow{time} (s,\Theta') \Leftrightarrow$$
$$\Theta' > \Theta \;\wedge\; (\forall \theta < \Theta'; t \in T : (s,\theta) \not\xrightarrow{t})$$

Assignment

`o <-` E `delay` T

at simulation time Θ creates a token with value E and timestamp $\Theta + T$.

In system definition stores must be initialized, except special stores `in user`, `in system`.

User stores: user responsible for initialization.
System stores:

```
time: real in system
```

gives simulation time Θ.

```
random: real in system
```

gives a random number in interval $0 \ldots 1$. Updated after each draw.

Exercise:

Model a system consisting of 3 subsystems:

`Generate` creates an object every time unit. The object should contain its time of creation as value.

`Process` receives objects one at a time. With equal probability they are either output or delayed 0.9 units and then tested again. After an output a new object can be processed.

`Report` receives finished objects and calculates the time they spent in the system. This value is reported in a `user` channel.

Top level:

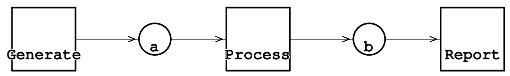

fig.16

`Generate` creates objects with current time in it. Channel `i` has an initial token and `gen` puts it back with `delay 1.0`.

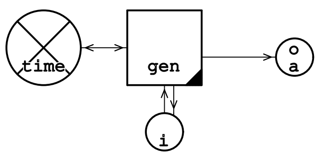

fig.17

`Process`: `copy` copies the object to an internal place. The body of `iter` is

```
if r < 0.5 then i <- i delay 0.9
else b <- i, e <- i fi
```

The `e` channel has an initial token. Inhibition of `copy` by `i` does not work here!

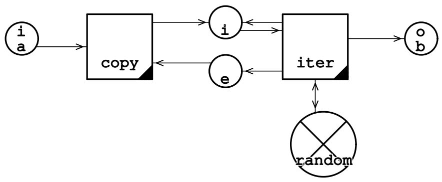

fig.18

`Report` subtracts gentime from curtime.

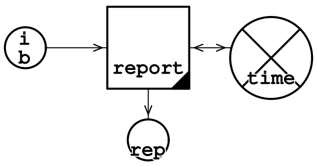

fig.19

Modularity

ExSpect specification divided in *modules*.
module contains *declarations* and *definitions*

declarations: definitions imported from other modules.
example:

```
random[x:real]: real in stat
union[x:$$T]: $T in utils
```

definitions can be *exported*.
module "utils" contains e.g. the definition

```
export union[x:$$T] :=
  if x={} then {}
  else pick(x) union
       union(pick(x) del x) fi: $T
```

definitions without "export" only applicable within module.

macro for inclusion of files:

```
include 'utils.h'
```

imports all declarations from module utils

The ".h" files can be generated automatically by the type checker.

Definitions can contain "where parts": local definitions.
example:

```
proc amt[in x:ord, store s:num |
   pre np > 0 and pp > 0] :=
   s <- s + np*pp
where np := x@amt: num
      pp := price(x@prod): num
```


Where parts also usefull for speedup recursive function definitions.

Example: Fibonnaci:

```
fib[x:num] :=
if x<2 then 1
else fib(x-1) + fib(x-2) fi: num
```

Better:

```
fib[x:num] := f(x,1,0): num
where f[x:num,y:num,z:num] :=
if x<2 then y+z
else f(x-1,y+z,y) fi: num
```

Exercise: define a function `freqcount[x:str,y:str]:num` that counts the number of times y occurs in x.
Examples:
`freqcount('Barbara','ar')` = 2,
`freqcount('aaaa','aa')` = 3.

Answer:

```
freqcount[x:str,y:str] :=
 if x = '' then 0
 else h(x,y) + freqcount(tail(x),y) fi:  num
where
h[x:str,y:str] :=
 if y = '' then 1
 elif x = '' or head(x) != head(y) then 0
 else h(tail(x),tail(y)) fi:   num
```


Examples

Some programming constructions in ExSpect:

Sequential composition

(Static part: `V(U(T(x)))`)

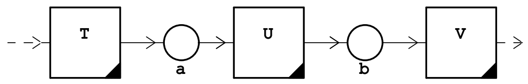

fig.20

Choice (conflict)
(Static part: `if..then..else..fi`)

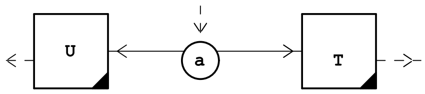

fig.21

preconditions may be added

Iteration, for, while
(Static part: quantors, recursion.)

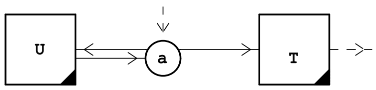

fig.22

Parallellism

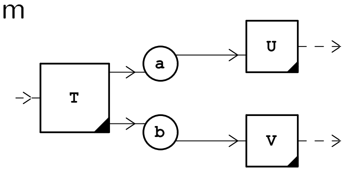

fig.23

Synchronization

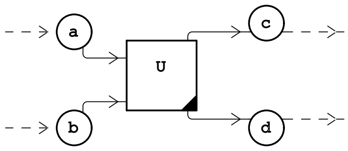

fig.24

Alternative via token passing

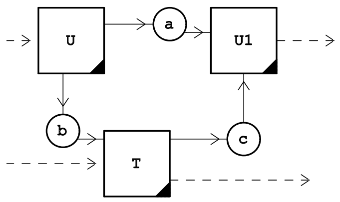

fig.25

Asynchronous communication

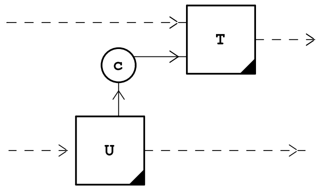

fig.26

Mutual exclusion (resource)

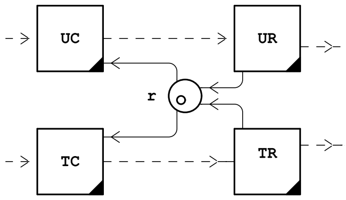

fig.27

Buffer with capacity k: free slots are modeled as resources.

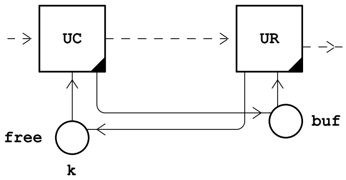

fig.28

Initial marking of free with k tokens.

Alternative with a store `S` of type `num><$T`.
initialization: `<<k,{}>>`

UC pre `pi1(S) > 0`
body `s <- <<pi1(S)-1, t ins pi2(S)>>`

UR pre `t elt pi2(S)`
body `s <- <<pi1(S)+1, t del pi2(S)>>`

Advantage of store approach: flexibility, Disadvantage: no invariants. If both needed: combination.