Start of Lecture 10: RSL: Imperative & Process Specs.

1.6. Imperative Constructs

1.6.1. Statements and State Changes

Unit
value
stmt: **Unit** → **Unit**
stmt()

- The **Unit** clause, in a sense, denotes "an underlying state"
  - which we, for simplicity, can consider as
  - a mapping from identifiers of declared variables into their values.
- Statements accept no arguments and, usually, operate on the state
  - through "reading" the value(s) of declared variables and
  - through "writing", i.e., assigning values to such declared variables.
- Statement execution thus changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Affixing () as an "only" arguments to a function "means" that () is an argument of type **Unit**.

1.6.2. Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

1.6.3. Statement Sequences and skip

2. **skip**
3. stm_1;stm_2;...;stm_n

1.6.4. Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

1.6.5. Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

1.6.6. Iterative Sequencing

8. **for** i **in** list · P(list(i)) **do** S(list(i)) **end**
9. **for** e **in** set · P(e) **do** S(e) **end**

1.7. Process Constructs
1.7.1. Process Channels

- Let A, B and C stand for three types of (channel) messages
- and i:IIdx, j:JIdx for channel array indexes, then:

channel
 c:A
channel
 {k[i]|i:IIdx}:B
 {ch[i,j]i:IIdx,j:JIdx}:C

Example 17 .... Modelling Connected Links and Hubs:

- Examples (17–20) are building up a model of one form of meaning of a transport net.
  - We model the movement of vehicles around hubs and links.
  - We think of each hub, each link and each vehicle to be a process.
  - These processes communicate via channels.

- We assume a net, $n : N$, and a set, vs, of vehicles.
- Each vehicle can potentially interact
  - with each hub and
  - with each link.
- Array channel indices *(vi,hi):IVH* and *(vi,li):IVL* serve to effect these interactions.
- Each hub can interact with each of its connected links and indices *(hi,li):IHL* serves these interactions.

type
 N, V, VI
value
 n:N, vs:V-**set**
 obs_VI: V → VI
type
 H, L, HI, LI, M
 IVH = VI×HI, IVL = VI×LI, IHL = HI×LI

- We need some auxiliary quantities in order to be able to express subsequent channel declarations.
- Given that we assume a net, $n : N$ and a set of vehicles, $vs : VS$, we can now define the following (global) values:
  - the sets of hubs, hs, and links, ls of the net;
  - the set, $ivhs$, of indices between vehicles and hubs,
  - the set, $ivls$, of indices between vehicles and links, and
  - the set, $ihls$, of indices between hubs and links.

value
 hs:H-**set** = obs_Hs(n), ls:L-**set** = obs_Ls(n)
 his:HI-**set** = {obs_HI(h)|h:H•h ∈ hs}, lis:LI-**set** = {obs_LI(h)|l:L•l ∈ ls},
 ivhs:IVH-**set** = {(obs_VI(v),obs_HI(h))|v:V,h:H•v ∈ vs∧h ∈ hs}
 ivls:IVL-**set** = {(obs_VI(v),obs_LI(l))|v:V,l:L•v ∈ vs∧l ∈ ls}
 ihls:IHL-**set** = {(hi,li)|h:H,(hi,li):IHL• h ∈ hs∧hi=obs_HI(h)∧li ∈ obs_LIs(h)}

- We are now ready to declare the channels:
  - a set of channels, {vh[i]|i:IVH·i∈ivhs} between vehicles and all potentially traversable hubs;
  - a set of channels, {vh[i]|i:IVH·i∈ivhs} between vehicles and all potentially traversable links; and
  - a set of channels, {hl[i]|i:IHL·i∈ihls}, between hubs and connected links.

channel
  {vh[i] | i:IVH · i ∈ ivhs} : M
  {vl[i] | i:IVL · i ∈ ivls} : M
  {hl[i] | i:IHL · i ∈ ihls} : M

. . . . **End of Example 17**

1.7.2. Process Definitions

- A process definition is a function definition.
- The below signatures are just examples.
- They emphasise that process functions must somehow express,
  - in their signature,
- via which channels they wish to engage in input and output events.
- Processes P and Q are to interact, and to do so “ad infinitum”.
- Processes R and S are to interact, and to do so “once”, and then yielding B, respectively D values.


```
value
  P: Unit → in c out {k[i]|i:IIdx} Unit
  Q: i:KIdx →  out c in k[i] Unit

  P()  ≡ ... c ? ... k[i] ! e ... ; P()
  Q(i) ≡ ...  c ! e ... k[i] ? ... ; Q(i)
```

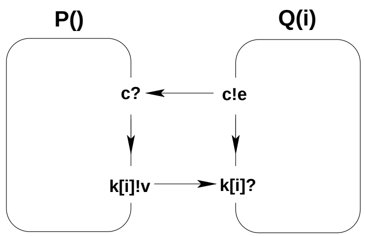

Figure 9: The P —— Q Process

Example 18 . . . . . . Communicating Hubs, Links and Vehicles:

- Hubs interact with links and vehicles:
  - with all immediately adjacent links,
  - and with potentially all vehicles.
- Links interact with hubs and vehicles:
  - with both adjacent hubs,
  - and with potentially all vehicles.
- Vehicles interact with hubs and links:
  - with potentially all hubs.
  - and with potentially all links.

value

hub: hi:HI × h:H → **in**,**out** {hl[(hi,li)|li:LI·li ∈ obs_LIs(h)]}
 in,**out** {vh[(vi,hi)|vi:VI·vi ∈ vis]} **Unit**
link: li:LI × l:L → **in**,**out** {hl[(hi,li)|hi:HI·hi ∈ obs_HIs(l)]}
 in,**out** {vh[(vi,li)|vi:VI·vi ∈ vis]} **Unit**
vehicle: vi:VI → (Pos × Net) → v:V →
 in,**out** {vh[(vi,hi)|hi:HI·hi ∈ his]}
 in,**out** {vl[(vi,li)|li:LI·li ∈ lis]} **Unit**

. . . . **End of Example 18**

1.7.3. Process Composition

- Let P and Q stand for names of process functions,
- i.e., of functions which express willingness to engage in input and/or output events,
- thereby communicating over declared channels.
- Let $\mathcal{P}$ and $\mathcal{Q}$ stand for process expressions,
- and let $\mathcal{P}_i$ stand for an indexed process expression, then:

$\mathcal{P} \parallel \mathcal{Q}$	Parallel composition
$\mathcal{P} \mathbin{[\,]} \mathcal{Q}$	Nondeterministic external choice (either/or)
$\mathcal{P} \sqcap \mathcal{Q}$	Nondeterministic internal choice (either/or)
$\mathcal{P} \between \mathcal{Q}$	Interlock parallel composition
$\mathcal{O}\ \{\ \mathcal{P}_i \mid \text{i:Idx}\ \}$	Distributed composition, $\mathcal{O} = \parallel, [\,], \sqcap, \between$

Example 19 . . . . **Modelling Transport Nets:**

- The net, with vehicles, potential or actual, is now considered a process.
- It is the parallel composition of
  - all hub processes,
  - all link processes and
  - all vehicle processes.

value

net: N → V-**set** → **Unit**
net(n)(vs) ≡
 ‖ {hub(obs_HI(h))(h)|h:H·h ∈ obs_Hs(n)} ‖
 ‖ {link(obs_LI(l))(l)|l:L·l ∈ obs_Ls(n)} ‖
 ‖ {vehicle(obs_VI(v))(obs_PN(v))(v)|v:V·v ∈ vs}

obs_PN: V → (Pos×Net)

- We illustrate a schematic definition of simplified hub processes.
- The hub process alternates, internally non-deterministically, ⌈⌉, between three sub-processes
  - a sub-process which serves the link-hub connections,
  - a sub-process which serves thos vehicles which communicate that they somehow wish to enter or leave (or do something else with respect to) the hub, and
  - a sub-process which serves the hub itself — whatever that is !

hub(hi)(h) ≡
 [] {**let** m = hl[(hi,li)] ? **in** hub(hi)($\mathcal{E}_{h_\ell}$(li)(m)(h)) **end**|i:LI·li ∈ obs_LI(h)}
⌈⌉ [] {**let** m = vh[(vi,hi)] ? **in** hub(vi)($\mathcal{E}_{h_v}$(vi)(m)(h)) **end**|vi:VI·vi ∈ vis}
⌈⌉ hub(hi)($\mathcal{E}_{h_{own}}$(h))

- The three auxiliary processes:
  - $\mathcal{E}_{h_\ell}$ update the hub with respect to (wrt.) connected link, *li*, information *m*,
  - $\mathcal{E}_{h_v}$ update the hub with wrt. vehicle, *vi*, information *m*,
  - $\mathcal{E}_{h_{own}}$ update the hub with wrt. whatever the hub so decides. An example could be signalling dependent on previous link-to-hub communicated information, say about traffic density.

$\mathcal{E}_{h_\ell}$: LI → M → H → H
$\mathcal{E}_{h_v}$: VI → M → H → H
$\mathcal{E}_{h_{own}}$: H → H

- The student is encouraged to sketch/define similarly schematic link and vehicle processes.

.... **End of Example 19**

1.7.4. Input/Output Events

- Let c and k[i] designate channels of type A
- and e expression values of type A, then:

[1] c?, k[i]?	input A value
[2] c!e, k[i]!e	output A value

value

[3] P: ... → **out** c ..., P(...) ≡ ... c!e ...	offer an A value,
[4] Q: ... → **in** c ..., Q(...) ≡ ... c? ...	accept an A value
[5] S: ... → ..., S(...) = P(...)‖Q(...)	synchronise and communicate

- [5] expresses the willingness of a process to engage in an event that
  - [1,3] "reads" an input, respectively
  - [2,4] "writes" an output.

Example 20 .... **Modelling Vehicle Movements:**

- Whereas hubs and links are modelled as basically static, passive, that is, inert, processes we shall consider vehicles to be "highly" dynamic, active processes.
- We assume that a vehicle possesses knowledge about the road net.
  - The road net is here abstracted as an awareness of
  - which links, by their link identifiers,
  - are connected to any given hub, designated by its hub identifier,
  - the length of the link,
  - and the hub to which the link is connected "at the other end", also by its hub identifier

- A vehicle is further modelled by its current position on the net in terms of either hub or link positions
  - designated by appropriate identifiers
  - and, when "on a link" "how far down the link", by a measure of a fraction of the total length of the link, the vehicle has progressed.

type
Net = HI $\overrightarrow{m}$ (LI $\overrightarrow{m}$ HI)
Pos = atH | onL
atH == mk_atH(hi:HI)
onL == mk_onL(fhi:HI,li:LI,f:F,thi:HI)
F = {|f:**Real**·0≤f≤1|}

- We first assume that the vehicle is at a hub.
- There are now two possibilities (1–2] versus [4–8]).
  - Either the vehicle remains at that hub
    - [1] which is expressed by some non-deterministic *wait*
    - [2] followed by a resumption of being that vehicle at that location.
  - [3] Or the vehicle (driver) decides to "move on":
    - [5] Onto a link, *li*,
    - [4] among the links, *lis*, emanating from the hub,
    - [6] and towards a next hub, *hi′*.
  - [4,6] The *lis* and *hi′* quantities are obtained from the vehicles own knowledge of the net.
  - [7] The hub and the chosen link are notified by the vehicle of its leaving the hub and entering the link,
  - [8] whereupon the vehicle resumes its being a vehicle at the initial location on the chosen link.

- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

```
type
  M == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(mk_atH(hi),net)(v) ≡
  [1]  (wait ;
  [2]   vehicle(vi)(mk_atH(hi),net)(v))
  [3]  ⌈⌉
  [4]  (let lis=dom net(hi) in
  [5]   let li:LI•li ∈ lis in
  [6]   let hi′=(net(hi))(li) in
  [7]   (vh[ (vi,hi) ]!mk_H_L(hi,li)||vl[ (vi,li) ]!mk_H_L(hi,li));
  [8]   vehicle(vi)(mk_onL(hi,li,0,hi′),net)(v)
  [9]   end end end)
```


- We then assume that the vehicle is on a link and at a certain distance "down", *f*, that link.
- There are now two possibilities ([1–2] versus [4–7]).
  - Either the vehicle remains at that hub
    - [1′] which is expressed by some non-deterministic *wait*
    - [2′] followed by a resumption of being that vehicle at that location.
  - [3′] Or the vehicle (driver) decides to "move on".
  - [4′] Either
    - [5′] The vehicle is at the very end of the link and signals the link and the hub of its leaving the link and entering the hub,
    - [6′] whereupon the vehicle resumes its being a vehicle at hub *h′*.
  - [7′] or the vehicle moves further down, some non-zero fraction down the link.
- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).


```
type
  M == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  δ:Real = move(h,f) axiom 0<δ≪1
  vehicle(vi)( mk_onL(hi,li,f,hi′),net)(v) ≡
  [1′]  (wait ;
  [2′]   vehicle(vi)(mk_onL(hi,li,f,hi′),net)(v))
  [3′]  ⌈⌉
  [4′]  (case f of
  [5′]     1 → ((vl[ vi,hi′ ]!mk_L_H(li,hi′)||vh[ vi,li ]!mk_L_H(li,hi′));
  [6′]           vehicle(vi)(mk_atH(hi′),net)(v)),
  [7′]     _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
  [8′]   end)
  move: H × F → F
```

. . . . **End of Example 20**

1.8. Simple RSL Specifications

- Besides the above constructs RSL also possesses module-oriented
  - scheme,
  - class and
  - object

  constructs.
- We shall not cover these here.
- An RSL specification is then simply
  - a sequence of one or more clusters of
    * zero, one or more sort and/or type definitions,
    * zero, one or more variable declarations,
    * zero, one or more channel declarations,
    * zero, one or more value definitions (including functions) and
    * zero, one or more and axioms.
- We can illustrate these specification components schematically:

type
  A, B, C, D, E, F, G
  Hf = A-**set**, Hi = A-**infset**
  J = B×C×...×D
  Kf = E^*, Ki = E^ω
  L = F $\overrightarrow{m}$ G
  Mt = J → Kf, Mp = J $\stackrel{\sim}{\to}$ Ki
  N == alpha | beta | ... | omega
  O == mk_Hf(as:Hf)
     | mk_Kf(el:Kf) | ...
  P = Hf | Kf | L | ...
variable
  vhf:Hf := ⟨⟩
channel
  chf:F, chg:G, {chb[i]|i:A}:B

value
  va:A, vb:B, ..., ve:E
  f1: A → B, f2: C $\stackrel{\sim}{\to}$ D
  f1(a) ≡ $\mathcal{E}_{f1}$(a)
  f2: E → **in**|**out** chf F
  f2(e) ≡ $\mathcal{E}_{f2}$(e)
  f3: **Unit** → **in** chf **out** chg **Unit**
  ...
axiom
  $\mathcal{P}_i$(f1,va),
  $\mathcal{P}_j$(f2,vb),
  ...
  $\mathcal{P}_k$(f3,ve)

- The ordering of these clauses is immaterial.
- Intuitively the meaning of these definitions and declarations are the following.
  - The **type** clause introduces a number of user-defined type names;
    * the type names are visible anywhere in the specification;
    * and either denote sorts or concrete types.
  - The **variable** clause declares some variable names;
    * a variable name denote some value of decalred type;
    * the variable names are visible anywhere in the specification:
      · assigned to ('written') or
      · values 'read'.
  - The **channel** clause declares some channel names;
    * either simple channels or arrays of channels of some type;
    * the channel names are visible anywhere in the specification.

- The **value** clause bind (constant) values to value names.
  * These value names are visible anywhere in the specification.
  * The specification

    type
      A
    value
      a:A

  * non-deterministically binds a to a value of type A.
  * Thuis includes, for example

    type
      A, B
    value
      f: A → B

  * which non-deterministically binds f to a function value of type A→B.

Example 21 . . . . A Neat Little "System":

- We present a self-contained specification of a simple system:
  - The system models
    - vehicles moving along a net, *vehicle*,
    - the recording of vehicles entering links, *enter_sensor*,
    - the recording of vehicles leaving links, *leave_sensor*, and
    - the *road_pricing payment* of a vehicle having traversed (*entered* and *left*) a link.
  - Note
    - that vehicles only pay when completing a link traversal;
    - that 'road pricing' only commences once a vehicle enters the first link after possibly having left an earlier link (and hub); and
    - that no *road_pricing payment* is imposed on vehicles entering, staying-in (or at) and leaving hubs.

  - We assume the following:
    - that each *link* is somehow associated with two pairs of *sensors:*
      - a pair of *enter* and *leave sensors* at one end, and
      - a pair of *enter* and *leave sensors* at the other end;

      and
    - a *road pricing* process
      - which records pairs of link enterings and leavings,
      - first one, then, after any time interval, the other,
      - with leavings leading to debiting of traversal fees;
- Our first specification
  - define types,
  - assume a net value,
  - declares channels and
  - state signatures of all processes.

- *ves* stand for vehicle entering (link) sensor channels,
- *vls* stand for vehicle leaving (link) sensor channels,
- *rp* stand for 'road pricing' channel
- *enter_sensor(hi,li)* stand for vehicle entering [sensor] process from hub *hi* to link *(*li).
- *leave_sensor(li,hi)* stand for vehicle leaving [sensor] process from link *li* to hub *(*hi).
- *road_pricing()* stand for the unique 'road pricing' process.
- *vehicle(vi)(...)* stand for the vehicle *vi* process.


```
type
  N, H, HI, LI, VI
  RPM == mk_Enter_L(vi:VI,li:LI) | mk_Leave_L(vi:VI,li:LI)
value
  n:N
channel
  {ves[obs_HI(h),li]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
  {vls[li,obs_HI(h)]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
  rp:RPM
type
  Fee, Bal
  LVS = LI →m VI-set,  FEE = LI →m Fee,  ACC = VI →m Bal
value
  link: (li:LI × L) → Unit
  enter_sensor: (hi:HI × li:LI) → in ves[hi,li],out rp Unit
  leave_sensor: (li:LI × hi:HI) → in vls[li,hi],out rp Unit
  road_pricing: (LVS×FEE×ACC) → in rp Unit
```

- To understand the sensor behaviours let us review the vehicle behaviour.
- In the *vehicle* behaviour defined in Example 20, in two parts, Slide 381 and Slide 383 we focus on the events
  - [7] where the vehicle enters a link, respectively
  - [5′] where the vehicle leaves a link.
- These are summarised in the schematic reproduction of the vehicle behaviour description.
  - We redirect the interactions between vehicles and links to become
  - interactions between vehicles and enter and leave sensors.

value

```
δ:Real = move(h,f) axiom 0<δ≪1
move: H × F → F
```



```
vehicle: VI → (Pos × Net) → V → Unit
vehicle(vi)(pos,net)(v) ≡
[1] (wait ;
[2]  vehicle(vi)(pos,net)(v))
[3]  ⌈⌉
   case pos of
     mk_atH(hi) →
[4–6]  (let lis=dom net(hi) in let li:LI•li ∈ lis in let hi′=(net(hi))(li) in
[7]      ves[ hi,li ]!vi;
[8]      vehicle(vi)(mk_onL(hi,li,0,hi′),net)(v)
[9]      end end end)
     mk_onL(hi,li,f,hi′) →
[4′]     (case f of
[5′–6′]    1 → (vls[ li,hi ]!vi; vehicle(vi)(mk_atH(hi′),net)(v)),
[7′]       _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
[8′]     end)
   end
```


- As mentioned on Slide 389 *link* behaviours are associated with two pairs of sensors:
  - a pair of *enter* and *leave sensors* at one end, and
  - a pair of *enter* and *leave sensors* at the other end;

value

```
link(li)(l) ≡
  let {hi,hi′} = obs_HIs(l) in
  enter_sensor(hi,li) ∥ leave_sensor(li,hi) ∥
  enter_sensor(hi′,li) ∥ leave_sensor(li,hi′) end
enter_sensor(hi,li) ≡
  let vi = ves[ hi,li ]? in rp!mk_Enter_LI(vi,li); enter_sensor(hi,li) end
leave_sensor(li,hi) ≡
  let vi = ves[ li,hi ]? in rp!mk_Leave_LI(vi,li); enter_sensor(li,hi) end
```


- The *LVS* component of the *road_pricing* behaviour serves,
  - among other purposes that are not mentioned here,
  - to record whether the movement of a vehicles “originates” along a link or not.
- Otherwise we leave it to the student to carefully read the formulas.

value

```
payment: VI × LI → (ACC × FEE) → ACC
payment(vi,li)(fee,acc) ≡
  let bal′ = if vi ∈ dom acc then add(acc(vi),fee(li)) else fee(li) end
  in acc † [ vi ↦ bal′ ] end
add: Fee × Bal → Bal [ add fee to balance ]
```

```
road_pricing(lvs,fee,acc) ≡ in rp
  let m = rp? in
  case m of
    mk_Enter_LI(vi,li) →
        road_pricing(lvs†[ li↦lvs(li)∪{vi} ],fee,acc),
    mk_Leave_LI(vi,li) →
        let lvs′ = if vi ∈ lvs(li) then lvs†[ li↦lvs(li)\{vi} ] else lvs end,
             acc′ = payment(vi,li)(fee,acc) in
        road_pricing(lvs′,fee,acc′)
  end end end
```

. . . . **End of Example 21**

End of Lecture 10: RSL: Imperative & Process Specs.