Topic 65

Concurrent Specification Programming

- The **prerequisites** for following this (part of the) lecture are that you have understood most, if not all, of what has been covered in previous lectures *and* you are interested in modelling concurrent behaviours.
- The **aims** are
  - ⋆ to motivate and introduce both simple `CSP` and the `RAISE` version of `CSP`, `RSL/CSP`, and
  - ⋆ to show a number of principles and techniques for modelling concurrent behaviours using `RSL`.

- The **objective** is
  - ⋆ to set the course student firmly on the road to modelling concurrent systems such as distributed systems, client/server systems, etc.
- The **treatment** is semiformal.

- In this lecture cluster we introduce a notation for expressing parallelism (also called concurrency):
  - ⋆ First we present a pure notation, a formal language, `CSP`: *Communicating Sequential Processes*.
  - ⋆ Then we present this notation's embedding in `RSL`.

Characterisation 21.85 By *concurrent programming* we shall understand programming with processes as a central notion: where processes are combined (in parallel) to form concurrent processes, where synchronous or asynchronous interaction between processes can be specified, and so on. ■

Characterisation 21.86 By *parallel programming* we mean the same as *concurrent programming*. ■

Characterisation 21.87 By *concurrent specification programming* we shall understand an abstract, property-oriented form of *concurrent programming*, one in which (relative or absolute) progress of processes is left unspecified, where choice between actions of proceses can be left unspecified (i.e., nondeterministic), and in which we deploy abstract types, and so on. ■

- In a separate series of lectures we shall introduce three sets of predominantly graphical notations:
  - ⋆ *Petri Nets*,
  - ⋆ *Message Sequence Charts* and *Live Sequence Charts*, and
  - ⋆ *Statecharts*.
- In this chapter we bring in a number of principles and techniques for modelling concurrent behaviours and the interaction between behaviours.
- We do so in a number of steps:
  - ⋆ First we informally examine some basic notions of behaviours.

- ⋆ Then, on a per intuition basis, we present some behaviour scenarios and show their possible formalisation using RSL's CSP sublanguage. But this sublanguage is not formally introduced.
- ⋆ Following that we present the "bare bones" of CSP.
- ⋆ After all these preliminaries we introduce,
  - ◇ more systematically, the RSL/CSP sublanguage,
  - ◇ and we suggest a calculus for transforming applicative, respectively imperative, RSL specifications into RSL/CSP specifications.
- ⋆ In another lecture series we cover extensions to the RSL/CSP sublanguage. These allow us to deal with "real time" and with time durations.

- Throughout we present many examples.

Behaviour and Process Abstractions

Characterisation 21.88 *Behaviour* is defined in Merriam–Webster's Collegiate Dictionary

- *(i) the manner of conducting oneself,*
- *(ii) anything that an organism does involving action and response to stimulation,*
- *(iii) the response of an individual, group, or species to its environment,*
- *(iv) the way in which someone behaves, an instance of such behavior,*
- *(v) the way in which something functions or operates*

. ■

- By behaviour we shall understand the organism to be anything spanning from a human, via any phenomena in “Mother Nature”, to an interpreter or a machine or a computer.

Characterisation 21.89 Merriam–Webster defines *process* as

- *(i) a natural phenomenon marked by gradual changes that lead toward a particular result,*
- *(ii) a natural continuing activity or function,*
- *(iii) a series of actions or operations conducing to an end, or more particularly*
- *(iv) a continuous operation or treatment especially in manufacture*

. ■

- We shall, more or less, take the two terms, ‘behaviour’ and ‘process’, as being synonymous.
  - ⋆ The only difference is a pragmatic one:
  - ⋆ When we use the term ‘behaviour’ we refer to an as yet unanalysed, hence not yet formalised, but otherwise precisely described understanding of some actual-world phenomenon.
  - ⋆ And when we refer to the term ‘process’ we refer to an analysed, precisely narrated and/or formalised specification of a behaviour, typically as we expect it to be more or less implemented by computer.

Introduction

- Entities are the "things" we can point to: *bank accounts*, *trains*, *timetables*, *people*, *rail nets*, *etc.*.
- Entities can be subject to actions:
  - ⋆ queries concerning (i.e., observations of) their state, i.e., predicates and functions
    (viz.: *account balance*, *train speed*, *journey duration*, *etc.*);
    and also
  - ⋆ operations that possibly alter their states, i.e., generator functions
    (viz.: *deposit*, *accelerate*, *reschedule*, *etc.*).

⋆ Any particular entity can be seen from the point of view of the sequences of actions that apply to it
(viz.: *open account*, *alternation of one or more deposits into or one or more withdrawals from the account*, *ended by a close account*).

◇ Such a sequence of actions may, for certain actions in the sequence, involve two or more entities

◇ for which other action sequences are defined
(viz.: *transfer between accounts*, *running train according to timetable schedule*, *etc.*).

  - ⋆ We therefore see that action sequences may interact.
  - ⋆ In this section we shall investigate means for describing interaction sequences, or as they are also called, behaviours or processes.

- That is, we shall otherwise — with the above caveat in mind — in general, treat the two terms behaviour and process synonymously.

On Process and Other Abstractions

- In *abstraction* and in *modelling* we have at our disposal a number of *abstraction styles.*
- These are either property-oriented or model-oriented.
- Within the former we usually speak of *algebraic* or *axiomatic* abstractions.
- Within the latter we can distinguish between *denotation* abstraction and *computation* abstraction.

- In this section we shall introduce yet another form of abstraction and modelling: it is *operational* (as are computation models).
- Do not confuse operation abstraction with operational abstraction.
- In *operation abstraction* we abstract *individual* (usually basic, i.e., primitive) *operations* (i.e., functions and predicates) over abstracted entities.
- In *operational abstraction* we focus upon, but do not necessarily detail, specific *sequences of operations* of a system.

- *Denotation abstraction* was first introduced around 1970 in order to model the meaning of computer programs, typically of imperative languages. The *denotation* of a computer program is then seen as some mathematical function.
- *Denotation abstraction* can, however, also be applied to other than computing concepts.
- *Computation abstraction* was likewise first introduced around 1964 in order to model abstract executions of computer programs.
- The term *computation abstraction* emphasises the concept computation.

- In the actual world we may not think of some phenomena as computations, but rather as sequences of actions.
- In this case we prefer to use the term *operational abstraction* when modelling the sequence aspect of these sequences of actions.
- When seemingly independent, concurrently operating phenomena (i.e., processes) occasionally interact, and when we wish to model both the concurrency and the interaction, then we apply *process abstraction.*

- So *process abstraction* is a more general form of *operational abstraction.*
- Several tools and techniques are offered for the modelling of processes:
  - ⋆ The CSP-oriented techniques and tools.
  - ⋆ The Petri net-oriented techniques and tools.
  - ⋆ The statechart-oriented techniques and tools.
  - ⋆ The live sequence chart-oriented techniques and tools.
- In this cluster of one or more lectures we shall exclusively illustrate some process concepts using the CSP-approach — couched, however, in the RSL/CSP subset.

Topic 66
Intuition
Illustrative Rendezvous Scenarios

Example 21.165 *Four Rendezvous Scenarios:*

- *One sender, one receiver.*
  - ⋆ Two persons, P and Q, walk in opposite directions down a street, towards each other.
  - ⋆ One person, say P, carries a letter for the other person, Q.
  - ⋆ Some previous agreement, i.e., a protocol, has been established between the two persons that an exchange of a letter is to take place.[22]
  - ⋆ They walk, most likely, at different and, in any case, unpredictable speeds. The speeds may vary, and they may be zero.

[22]The compositional aspects of each of the four kinds of "rendezvous" classes of the diagrams of Fig. 21.26 and of the four corresponding formal specifications on Slides 1658 and 1662 "embody" this "agreement".

⋆ The letter deliverer and the letter receiver are willing to hand over, resp. to receive, i.e., to 'relay', the letter at any point.

⋆ As they are walking, the two persons are not performing any activities other than walking and being willing to 'relay'.

⋆ And as they meet — i.e., as they rendezvous — the delivering person "hands over" the letter which is simultaneously received by the receiving person.

⋆ After they have relayed the letter they both walk on in their respective directions.

⋆ If either P or Q refuses to walk, then the combined process fails, i.e., *deadlocks.*

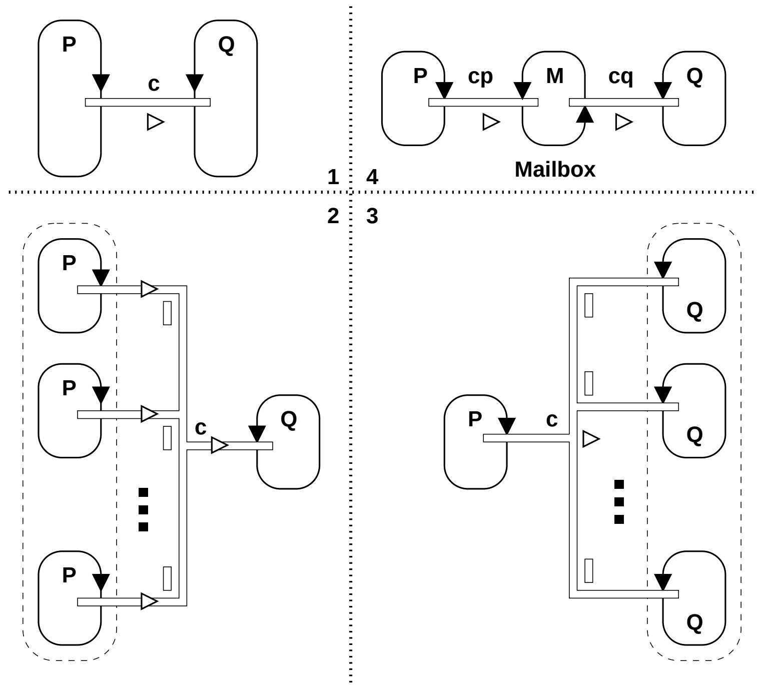


Figure 21.26: Four schematic “rendezvous” classes

Variants on the scenario above could be:

- *Any sender, one receiver.*
  - ⋆ The letter sender may be any one of a number of willing persons, P_1, P_2, ..., P_m,
  - ⋆ but the letter, and then at most one, is receivable only by a specific person Q, say standing still on the street.
  - ⋆ We consider P_1, P_2, ..., P_m and Q as processes.
  - ⋆ The various P_i are walking, each at their own speed, as was P in the previous scenario, but now in any direction, up or down, the street, and hence meeting Q sooner or later — with the first one so meeting delivering the letter.
- If either Q refuses, or all P_i refuse to walk, then the combined process fails, i.e., *deadlocks*.

- *One sender, any receiver.*
  - ⋆ The letter sender, P, is thought of as a fixed person, say standing still on the street,
  - ⋆ but the letter may be received by the "first" of a number of willing recipients, Q_1, Q_2, . . . , Q_n.
  - ⋆ We consider P, Q_1, Q_2, . . . , and Q_n as processes.
  - ⋆ The various Q_i are walking, each at their own speed, as was Q in the previous scenario, but now in any direction, up or down, the street, and hence meeting P sooner or later — with the first one so meeting receiving the letter.
  - ⋆ If either P refuses, or all Q_i refuse to walk, then the combined process fails, i.e., the combined process fails, or *deadlocks.*

- *Send/receive via a mailbox.*
  - ⋆ Letters are posted in an at most one letter capacity mailbox, M, by a sender, or any number of senders, and retrieved from that mailbox by a receiver, or any number of receivers.
  - ⋆ We consider P_1, P_2, ..., P_m Q_1, Q_2, ..., Q_n and M as processes.
- If no P_i puts a letter in the mailbox, then any Q_j attempting to fetch the letter *deadlocks.*

- Cyclic versions of the initial scenario and the three subsequent variations described above are illustrated in Fig. 21.26's respective four cases: 1–4.
- We will explain the graphics of Fig. 21.26.
- For simplicity and generality we have shown all processes as rounded-edge boxes with arrows.
- The thick line of the rounded-edge boxes is intended to designate a cyclic sequence of actions including the event-causing actions.
- The arrows are intended to show direction of execution (black) or communication (white).

- The horizontal bars "touching" ("overlapping") two or more boxes are intended to show synchronisation and communication rendezvous.
- The tiny rectangles (▯) along the rendezvous of parts **2** and **3** are intended to show nondeterministic choice as to from which of the (2) **P**'s (3, **Q**'s) **Q** (resp. **P**) will accept input.
- In part **4** the **mailbox** process **M** alternates between being ready to receive a letter from **P** and delivering that letter to **Q.**
- The four "box and arrow" diagrams of Fig. 21.26 correspond to the four sets of abstract process (i.e., function) definitions of the below Schematic "Rendezvous" Specifications 1–2–3 and Schematic "Rendezvous" Specification 4.

Schematic "Rendezvous" Specifications 1–2–3

```
type Info
channel c,cp,cq:Info
value
  P: Unit → out c Unit
  P() ≡ let i = write_letter() in c ! i end ; P()

  Q: Unit → in c Unit
  Q() ≡ let i = c ? in read_letter(i) end ; Q()

  write_letter: Unit → Info, read_letter: Info → Unit

  S1: Unit → Unit, S1() ≡ P() ∥ Q()
  S2: Nat Unit → Unit, S2(m) ≡ ∥ { P() | x:{1..m} } ∥ Q()
  S3: Nat Unit → Unit, S3(n) ≡ P()  ∥ (∥ { Q() | x:{1..n} })
```

- Process S1 is the parallel composition (‖) of processes P and Q.
- Process S2 is the parallel composition of process Q with the parallel, distributed composition of processes P, one for each index set 1..m.
- Process S3 is the parallel composition of process P with the parallel, distributed composition of processes Q, one for each index set 1..n.

- Info is the type of the information contained in the letter.
- c is what is known as a channel between P and Q in parts 1–3.
- Channels allow pairs of processes to share events;
- cp and cq are the channels between P and M, respectively between M and Q.
- P writes a letter, hands it over on channel c to Q — as prescribed by the output [!] / input [?] pair (c ! i,c ?).

- This is true in all parts 1–3.
- In part 1 it is simply so: Two processes (P and Q) share channel c, and thus share events.
- In part 2 many (m) processes P share the same channel c with one process Q. Q will not know which of the m P processes sent the letter to Q.
- In part 3 many (n) processes Q share same channel c with one process P. P will not know which of the n Q processes received the letter.
- All P and Q processes are cyclic: P produces letters, and Q consumes letters. They both cycle for each production, resp. consumption.

Schematic “Rendezvous” Specification 4

$\mathrm{S4}\colon \textbf{Unit} \to \textbf{Unit},\ \mathrm{S4}() \equiv \mathrm{P}'() \parallel \mathrm{M}() \parallel \mathrm{Q}'()$

$\mathrm{P}'\colon \textbf{Unit} \to \textbf{out}\ \mathrm{cp}\ \textbf{Unit}$
$\mathrm{P}'() \equiv \textbf{let}\ \mathrm{i} = \mathrm{write\_letter}()\ \textbf{in}\ \mathrm{cp}\ !\ \mathrm{i}\ \textbf{end}\ ;\ \mathrm{P}'()$

$\mathrm{M}\colon \textbf{Unit} \to \textbf{in}\ \mathrm{cp}\ \textbf{out}\ \mathrm{cq}\ \textbf{Unit}$
$\mathrm{M}() \equiv \textbf{let}\ \mathrm{i} = \mathrm{cp}\ ?\ \textbf{in}\ \mathrm{cq}\ !\ \mathrm{i}\ \textbf{end}\ ;\ \mathrm{M}()$

$\mathrm{Q}'\colon \textbf{Unit} \to \textbf{in}\ \mathrm{cq}\ \textbf{Unit}$
$\mathrm{Q}'() \equiv \textbf{let}\ \mathrm{i} = \mathrm{cq}\ ?\ \textbf{in}\ \mathrm{read\_letter}(\mathrm{i})\ \textbf{end}\ ;\ \mathrm{Q}'()$

- Process $\mathsf{P'}$ now sends (i.e., drops) the letter to (i.e., in) mailbox M. The relation between $\mathsf{P'}$ and M is as between P and Q in part 1.
- Process q now receives (i.e., fetches) the letter from mailbox M. The relation between M and $\mathsf{Q'}$ is as between P and Q in part 1.
- Process M is a one-item buffer, it alternates between receiving and sending. It recycles for each pair of receive–sends.

■

Diagram and Notation Summary

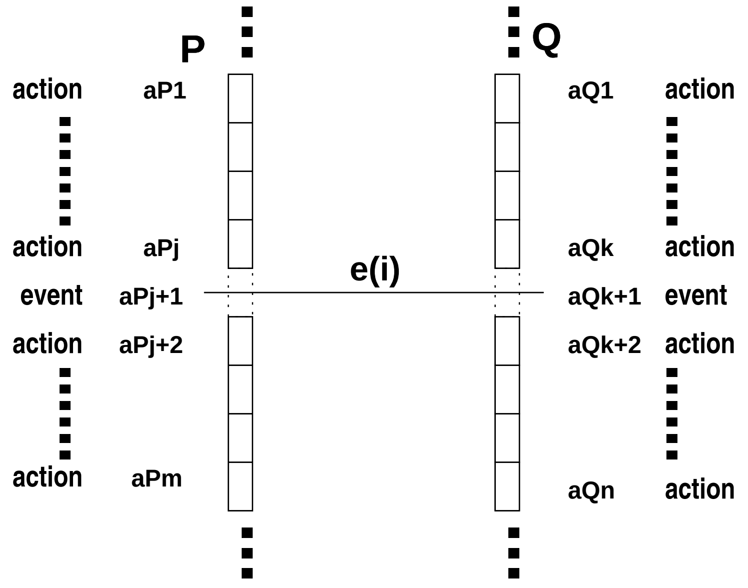
- So we have introduced the concepts of processes
- and their "rendezvous" output (!) and input (?) synchronisation and communication.
- We have sketched informal ways of picturing process structures (cf. Fig. 21.26);
- and we have informally shown formal notations (Figs. 21.165 and 21.165).
- Before going on to a more systematic, formal introduction of a (so-called "pure") notation for `[T]CSP` and the corresponding `CSP`-like notation for the process concepts of `RSL`,
- we will further review, illustrate and thus motivate the `CSP` process concepts.

On a Trace Semantics

- *Actions* change the data as well as the control state[23] and are thought of as taking place instantaneously, i.e., with no observable time duration.
- *Processes*, from one viewpoint, can be said to be sequences of actions.
- *Events* are phenomena that also take place instantaneously, but which, in and by themselves, do not change the data state, but (usually) cause actions to take place, i.e., be "triggered", thereby changing the control state.
- *Processes*, from another viewpoint, can be said to be sequences of events and — if causing actions — then possibly also sequences of actions.

[23] By a data state we understand "something" that records and remembers the values of various usually named data items, like a storage. By a control state we understand the interpreter's awareness of the point in a program cum specification text which is being interpreted by the interpreter.

- A process can exchange information with another process through what we shall call synchronised events (Fig. 21.27).
- *Systems* may consist of many processes synchronising on events and exchanging (communicating) information during such synchronised *'rendezvous'*.



Processes P and Q Rendezvous to cause Event e(i) after respective execution of Actions aPj, resp. aQk

Figure 21.27: Stylised “rendezvous” situation

- The *behaviour* of systems can, for example, be the set of sequences (traces) of externally observable events,
- or can, more generally, be the set of traces of both externally and internally observable events.
- For the conceptual example of Fig. 21.27 the system is that of the parallel ($\|$) combination of processes P and Q: $\mathsf{P}\|\mathsf{Q}$.
- The external behaviour is: $\{\langle\mathsf{e(i)}\rangle\}$.
- The internal behaviour — expressed, as above, in some metalinguistic notation — is:

$$\{\{\langle \text{ aP1, } \ldots, \text{ aPj } \rangle \bowtie \langle \text{ aQ1, } \ldots, \text{ aQk } \rangle\}$$
$$\hat{}\langle\{ \text{ aPj+1, e(i), aQk+1 } \}\rangle\hat{}$$
$$\{\langle \text{ aPj+2, } \ldots, \text{ aPm } \rangle \bowtie \langle \text{ aQk+2, } \ldots, \text{ aQn } \rangle\}\}.$$

Example 21.166 *Some Trace Semantics:*

type
  M
channel
  pq: M, qr: M
value
  S: **Unit** $\rightarrow$ **Unit**
  P: **Unit** $\rightarrow$ **out** pq **Unit**
  Q: **Unit** $\rightarrow$ **in** pq, **out** qr **Unit**
  R: **Unit** $\rightarrow$ **in** qr **Unit**

  S() $\equiv$ P() $\parallel$ Q() $\parallel$ R()

  P() $\equiv$ a1 ; pq!m ; a2 ; P()
  Q() $\equiv$ b1 ; **let** m = pq? **in** qr!m **end** ; b2; Q()
  R() $\equiv$ c1 ; qr ? ; c2 ; R()

- Traces observed of P, Q and R are:

$\mathcal{P}$: ⟨a1;pq!m;a2;a1;pq!m;a2;a1;pq!m;a2;a1;...⟩
$\mathcal{Q}$: ⟨b1;pq?;qr!m;b2;b1;pq?;qr!m;b2;b1;pq?;qr!m;b2;b1;...⟩
$\mathcal{R}$: ⟨c1;qr?;c2;c1;qr?;c2;c1;qr?;c2;c1;...⟩

- Traces potentially observable of S are:

$\mathcal{S}$: {⟨a1;b1;c1;{pq!m‖pq?};a2;{qr!m‖qr?};b2;{pq!m‖pq?};c2;...⟩,
⟨a1;b1;c1;{pq!m‖pq?};{qr!m‖qr?};a2;b2;{pq!m‖pq?};c2;...⟩,
⟨a1;b1;{pq!m‖pq?};c1;{qr!m‖qr?};a2;c2;b2;{pq!m‖pq?};...⟩,
... }

■

Some Characterisations: Processes, Etcetera

- One way of expressing the meaning of a process expression, is to express it as a set of traces of observable (output/input) events.

Characterisation 21.90 By a *process* we (semantically) mean a trace. ■

Characterisation 21.91 By a *process definition* we syntactically mean a function definition, and semantically a set of traces. ■

Characterisation 21.92 By *concurrent processes* we mean a set of two or more processes. ■

Characterisation 21.93 By the *global process environment* we mean the surroundings with which the process may interact, i.e., share in events, but excluding other defined and channel-connected processes. ∎

Characterisation 21.94 By a *process environment* we mean the set of other processes and the global surroundings from which the process may receive input and (or) to which it may deliver output, that is, with which the process may interact, i.e., share in events. ∎

Characterisation 21.95 By an *event* we mean a process event, the occurrence of an input (from the environment, including another process) or an output (to the environment, including another process) or both. The latter designates an internal event. ■

Characterisation 21.96 By an *externally observable process trace*, or just an *external trace*, we mean a sequence of process events. ■

Principle of Process Modelling

- So when do we choose to introduce processes into our models?
- The answer is not that straightforward.
- We can indeed model processes without introducing the explicit process (channel, output, input) notation so far informally illustrated, for example, by nondeterministically defined transition functions over configurations that contain set- or map-oriented values whose elements model the control state of individual processes.

Principle 21.26 *Process Modelling:* We choose to model, in terms of processes and events, phenomena in the real world, i.e., “in some application domain”, or in computing, when we wish to emphasise concurrently interacting components, that is, how they synchronise and communicate. ■

Components ≡ Processes

- The concept of *component* is perhaps the one that we will rather assume for granted. However:

Characterisation 21.97 By a *component* we shall loosely understand a structured set of [variable or constant] (i) values [modelling certain nouns], (ii) predicates, [observer] functions and [generator] operations over these [modelling certain verbs], (iii) and events that stand for willingness to “communicate”, i.e., to accept and/or present values to other components, including that of an “outside” [external] world. ■

- In this sense a component becomes synonymous with what we shall now call a process.
- The concept of 'object', as in object-oriented programming , is sometimes used where we here use the term component — or "our" notion of 'component' is (then) a set of such objects (object modules).
- We shall, in another lecture series, elaborate more on object-oriented specification and the relationship between our concept of component (i.e., process or process definition) and that of the more commonly accepted use of objects and object-orientedness.
- Meanwhile, let us consider some component examples.

Informal Examples

Example 21.167 *Atomic Component — A Bank Account:* When we informally speak of the phenomena that can be observed in connection with a bank account, we may first bring up such things as:

- (i) The balance (or cash, a noun), the credit limit (noun), the interest rate (noun), the yield (noun); and
- (ii) the opening (verb) of, the deposit (verb) into, the withdrawal (verb) from and the closing (verb) of an account.
- Then we may identify (iii) the events that trigger the opening, deposit, withdrawal and closing actions.

- We may thus consider a bank account — with this structure of
  - ⋆ (i) values,
  - ⋆ (ii) actions, and
  - ⋆ (iii) ability to respond to external events

  — to be a component, i.e., a process.

■

- Components are either atomic or composite.
- In the latter case we can — often more or less arbitrarily — show a decomposition of a component into two or more subcomponents.

Example 21.168 *Composite Component — A Bank:*

- Likewise, continuing the above example, we can speak of a bank as consisting of any number of bank accounts,
- i.e., as a composite component of proper constituent bank account components.
- Other proper constituent components are:
  - ⋆ the customers (who own the accounts),
  - ⋆ the bank tellers (whether humans or machines) who services the accounts as instructed by customers,
  - ⋆ etc.

■

- In the above we have stressed the “internals” of the atomic components.
- When considering the composite components we may wish to emphasise the interaction between components.

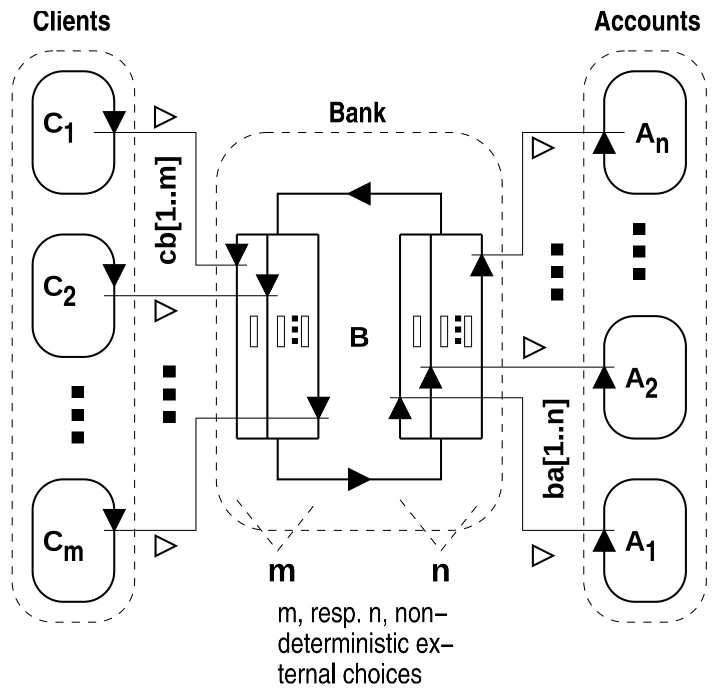


Figure 21.28: A fifth schematic “rendezvous” class

Example 21.169 *One-Way Composite Component Interaction:*

- A simple one-way client-to-account deposit.
  - ⋆ A customer may instruct
  - ⋆ a bank teller
  - ⋆ to deposit monies
  - ⋆ handed over from the customer to the bank teller
  - ⋆ into an appropriate account,

  and we see an interaction between three “atomic” components:
  - ⋆ the client(s),
  - ⋆ the bank teller(s) and
  - ⋆ the account(s).
- This scenario is very much like part 4 in Fig. 21.26, see also Fig. 21.28.

- shows a set of distinct client processes. A client may have one or more accounts and clients may share accounts.
- For each distinct account there is an account process.
- The bank (i.e., the bank teller) is a process. It is at any one time willing to input a cash-to-account (a,d) request from any client (c).
- There are as many channels into (out from) the bank process as there are distinct clients (resp. accounts).

type
  Cash, Cash, Cidx, Aidx
channel
  $\{$ cb[c]:(Aidx$\times$Cash) | c:Cidx $\}$
  $\{$ ba[a]:Cash | a:Aidx $\}$
value
  S5: **Unit** $\rightarrow$ **Unit**
  S5() $\equiv$ Clients() $\|$ B() $\|$ Accounts()

  Clients: **Unit** $\rightarrow$ **out** $\{$ cb[c] | c:Cidx $\}$ **Unit**
  Clients() $\equiv$ $\|$ $\{$ C(c) | c:Cidx $\}$

  C: c:Cidx $\rightarrow$ **out** cp[c] **Unit**
  C(c) $\equiv$ **let** (a,d):(Aidx$\times$Cash) = ... **in** cb[c] ! (a,d) **end** ; C(c)

type
  A_Bals = Aindex $\overrightarrow{m}$ Cash
value
  abals: A_Bals

  Accounts: **Unit** → **in** { ba[a] | a:AIndex } **Unit**
  Accounts() ≡ ‖ { A(a,abals(a)) | a:AIndex }

  A: a:Aindex × Balance → **in** ba[a] **Unit**
  A(a,d) ≡ **let** d′ = ba[a] ? **in** A(a,d+d′) **end**

  B: **Unit** → **in** { cb[c] | c:Cidx } **out** { ba[a] | a:Aidx } **Unit**
  B() ≡ ⌷ {**let** (a,d) = cb[c] ? **in** ba[a] ! d **end** | c:Cidx} ; B()

- We comment on the deposit example.
  - ⋆ With respect to the use of notation above,
    - ◇ there are Cindex client-to-bank channels, and Aindex bank-to-account channels.
    - ◇ The banking system (S5) consists of a number of concurrent processes: Cindex clients, Aindex accounts and one bank.
    - ◇ From each client process there is one output channel, and into each account process there is one input channel.
    - ◇ Each client and each account process cycles around depositing, respectively cashing monies.
    - ◇ The bank process is nondeterministically willing ($[]$) to engage in a rendezvous with any client process, and passes any such input onto the appropriate account.

- Generally speaking,
  - ⋆ we illustrated a banking system of many clients and many accounts.
  - ⋆ We only modelled the deposit behaviour from the client via the bank teller to the account.
  - ⋆ We did not model any reverse behaviour, for example, informing the client as to the new balance of the account.
- So the two bundles of channels were both one-way channels.
- We shall later show an example with two-way channels.

■

Example 21.170 *Multiple, Diverse Component Interaction:*

- We illustrate composite component interaction.
- At regular intervals, as instructed by some service scripts associated with several distinct kinds of accounts, transfers of monies may take place between these.
- For example, a regular repayment of a loan may involve the following components, operations and interactions:
  - ⋆ An appropriate repayment amount, p, is communicated from client k to the bank's script servicing component se (3).
  - ⋆ Based on the loan debt and its interest rate (d,ir) (4), and this repayment (p), a distribution of annuity (a), fee (f) and interest (i) is calculated.

- ⋆ The loan repayment sum total, p, is subtracted from the balance, b, of the demand/deposit account, dd_a, of the client (5).
- ⋆ A loan service fee, f, is added to the (loan service) fee account, f_a, of the bank (7).
- ⋆ The interest on the balance of the loan since the last repayment is added to the interest account, i_a, of the bank (8), and
- ⋆ the difference, a, (the effective repayment), between the repayment, p, and the sum of the fee and the interest is subtracted from the principal, p, of the mortgage account, m_a, of the client (6).

- In process modelling the above we are stressing the communications.
- As we shall see, the above can be formally modelled as below.

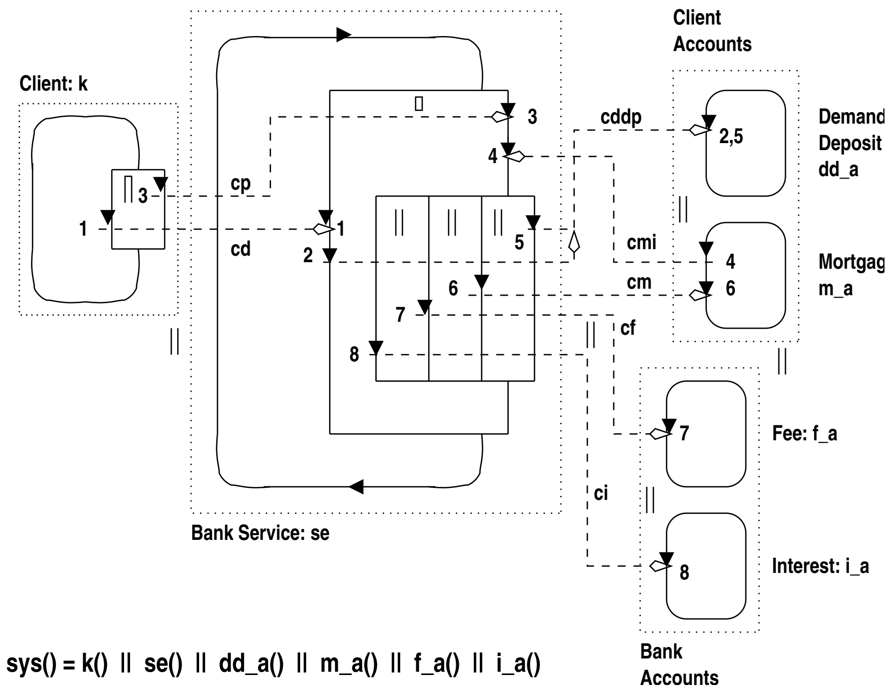


Figure 21.29: A loan repayment scenario

type
  Monies,Deposit,Loan,
  Interest_Income,Fee_Income = **Int**,
  Interest = Rat
channel
  cp,cd,cddp,cm,cf,ci:Monies, cmi:Interest
value
  sys: **Unit** $\rightarrow$ **Unit**,
  sys() $\equiv$ se() $\|$ k() $\|$ dd_a(b) $\|$ m_a(p) $\|$ f_a(f) $\|$ i_a(i)

```
k: Unit → out cp,cd  Unit
k() ≡
      (let p:Nat • /* p is some repayment, 1 */ in cp ! p end
    ⌈⌉
      let d:Nat • /* d is some deposit, 2 */ in cd ! d  end)
   ; k()

se: Unit → in cd,cp,cmi out cddp,cm,cf,ci  Unit
se() ≡
      ((let d = cd ? in cddp ! d end) /* 1,2 */
    ⌈⌉
      (let (p,(ir,ℓ)) = (cp ?,cmi ?) in /* 3,4 */
      let (a,f,iv) = o(p,ℓ,ir) in
      (cddp ! (−p) ∥ cm ! a ∥ cf ! f ∥ ci ! iv) end end)) /* 5,6,7,8 */
   ; se()
```

dd_a: Deposit → **in** cddp **Unit**
dd_a(b) ≡ dd_a(b + cddp ?) /* 2,5 */

m_a: Interest × Loan → **out** cmi **in** cm **Unit**
m_a(ir,ℓ) ≡ cmi ! (ir,ℓ) ; m_a(ir,ℓ− cm ?) /* 4;6 */

f_a: Fee_Income → **in** cf **Unit**
f_a(f) ≡ f_a(f + cf ?) /* 7 */

i_a: Interest Income → **in** ci **Unit**
i_a(i) ≡ i_a(i + ci ?) /* 8 */

© Dines Bjørner Fredsvej 11 DK-2840 Holte Denmark and © Springer Tiergartenstraße 17 D 69121 Heidelberg Germany

The formulas above express:

- The composite component, a bank, consists of:
  - ⋆ a customer, k, connected to the bank (service), se, via channels cd, cp
  - ⋆ that customer's demand/deposit account, dd_a, connected to the bank (service) via channels cdb, cddp
  - ⋆ that customer's mortgage account, m_a, connected to the bank (service) via channel cm
  - ⋆ a bank fees income account, f_a, connected to the bank (service) via channel cf
  - ⋆ a bank interest income account, i_a, connected to the bank (service) via channel ci

- The customer demand/deposit account is willing, at any time, to nondeterministically engage in communication with the service: either accepting (?) a deposit or loan repayment (2 or 5), or delivering (!) information about the loan balance and interest rate (4).
- We model this "externally inflicted" behaviour by (what is called) the *external nondeterministic choice*, $\square$, operation.
- The service component, in a nondeterministic external choice, $\square$, either accepts a customer deposit (cd?) or a mortgage payment (cp?).
- The deposit is communicated (cddp!d) to the demand/deposit account component.

- The fee, interest and annuity payments are communicated in parallel (‖) to each of the respective accounts: bank fees income (cf!f), bank interest income (ci!i) and client mortgage (cm!a) account components.
- The customer is unpredictable, may issue either a deposit or a repayment interaction with the bank.
- We model this "self-inflicted" behaviour by (what is called) the *internal nondeterministic choice*, $\bigcap$, operation.

■

[21] See the definition of what is meant by nondeterministic external choice right after this example.
[22] See the definition of what is meant by nondeterministic internal choice right after this example.

Characterisation 21.98 By a *nondeterministic external choice* we mean a nondeterministic decision which is effected, not by actions prescribed by the text in which the $\llbracket$ operator occurs, but by actions in other processes. That is, speaking operationally, the process honouring the $\llbracket$ operation does so by "listening" to the environment. ■

Characterisation 21.99 By *nondeterministic internal choice* we mean a nondeterministic decision that is implied by the text in which the $\lceil\rceil$ operator occurs. Speaking operationally, the decision is taken locally by the process itself, not as the result of any event in its surroundings. ■

Some Modelling Comments — An Aside

- Examples 21.169 and 21.170 illustrated one-way communication, from clients via the bank to accounts.
- Example 21.169 illustrated bank "multiplexing" between several (m) clients and several (n) accounts.
- Example 21.170 illustrated a bank with just one client and one pair of client demand/deposit and mortgage accounts.
- Needless to say, a more realistic banking system would combine the above.
- Also, we have here chosen to model each account as a process.

  - ⋆ It is reasonable to model each client as a separate process, in that the collection of all clients can be seen as a set of independently and concurrently operating components.
  - ⋆ To model the large set of all accounts as a similarly large set of seemingly independent and concurrent processes can perhaps be considered a "trick": It makes, we believe, the banking system operation more transparent.

- In the next example we augment the first example with an account balance response being sent back from the account via the bank to the client.

Examples Continued

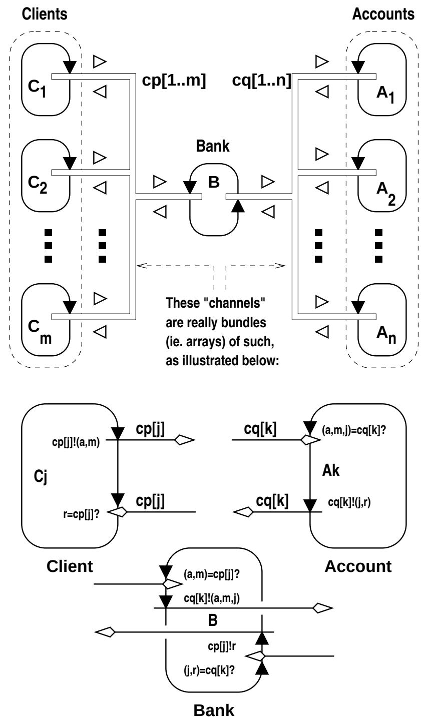


Figure 21.30: Two-way component interaction

Example 21.171 *Two-Way Component Interaction:*

- The present example “contains” that of the one-way component interaction of Example 21.169.
- Each of the client, bank and account process definitions are to be augmented as shown in Fig. 21.30 and in the formulas that follow
- (cf. Fig. 21.28 and the formulas in Example 21.169).

```
type
  Cash, Balance, CIndex, AIndex
  CtoB = AIndex × Cash,
  BtoC = Balance,
  BtoA = Cindex × Cash,
  AtoB = Cindex × Balance
channel
  cb[1..m] CtoB|BtoC, ba[1..n] BtoA|AtoB
value
  S6: Unit → Unit
  S6() ≡
    ‖ { C(c) | c:CIndex } ‖ B() ‖
    ‖ { A(a,b,r) | a:AIndex, b:Balance, r:Response • ... }
```

```
C: c:CIndex → out cp[ c ] Unit
C(c) ≡
  let (a,d):(AIndex×Cash) = ... in
  cb[ c ] ! (d,a) end let r = cb[ c ] ? in C(c) end

B: Unit → in,out {cb[ c ]|c:CIndex} in,out {ba[ a ]|a:AIndex} Unit
B() ≡ [] {let (d,a) = cb[ c ] ? in ba[ a ] ! (c,d) end | c:Cindex} []
      [] {let (c,b) = ba[ a ] ? in bc[ c ] ! b end | a:Aindex} ; B()

A: a:Aindex × Balance → in,out ba[ a ] Unit
A(a,b) ≡ let (c,m) = ba[ a ] ? in ba[ a ] ! (m+b) ; A(a,m+b) end
```

- We explain the formulas above.
  - ⋆ Both the C and the A definitions specify pairs of communications: deposit output followed by a response input, respectively a deposit input followed by a balance response output.
  - ⋆ Since many client deposits may occur while account deposit registrations take place, client identity is passed on to the account, which “returns” this identity to the bank — thus removing a need for the bank to keep track of client-to-account associations.
  - ⋆ The bank is thus willing, at any moment, to engage in any deposit and in any response communication from clients, respectively accounts.
  - ⋆ This is expressed using the nondeterministic external choice combinator $[\,]$.

■

Some System Channel Configurations

We have seen, so far, a number of configurations of channels and processes. Figure 21.31 attempts to diagram a few generic configurations of processes and channels. There may be channels between P, Q, P_j and Q_i processes and other (non-P, etc., and non-Q, etc.) processes, but they are not shown. We shall comment on each of these configurations:

[A] An event (a synchronisation and communication) between P and some Q_i prevents any other such event for the duration of the $P - Q_i$ event. Any other Q_j process (for $j \neq i$) may engage in other events with other processes, or own actions during the $P - Q_i$ event.

[**B**] An event (a synchronisation and communication) between P and Q_i prevents any other Q_j from engaging in an event with P for the duration of the $P - Q_i$ event. Any other Q_j process (for $j \neq i$) may engage in other events with other non-P processes, or own actions during the $P - Q_i$ event.

[C] An event (a synchronisation and communication) between some P_j and some Q_i prevents any other such $P_k - Q_\ell$ event for the duration of the $Pj - Qi$ event. Any other P_k and Q_ℓ processes (for $k \neq j$ and $j \neq i$) may engage in other events with other non-P and non-Q processes, or own actions during the $P_j - Q_i$ event.

[D] An event (a synchronisation and communication) between some P_j and some Q_i prevents any other $P_j - Q_k$ event for the duration of the Pj_Q_i event, but does not prevent a $P_\ell - Q_k$ event for $\ell \neq i$ and $k \neq j$. Etcetera. Please analyse other possible process engagements yourself!

[E] Etcetera. Please analyse the diagram yourself!

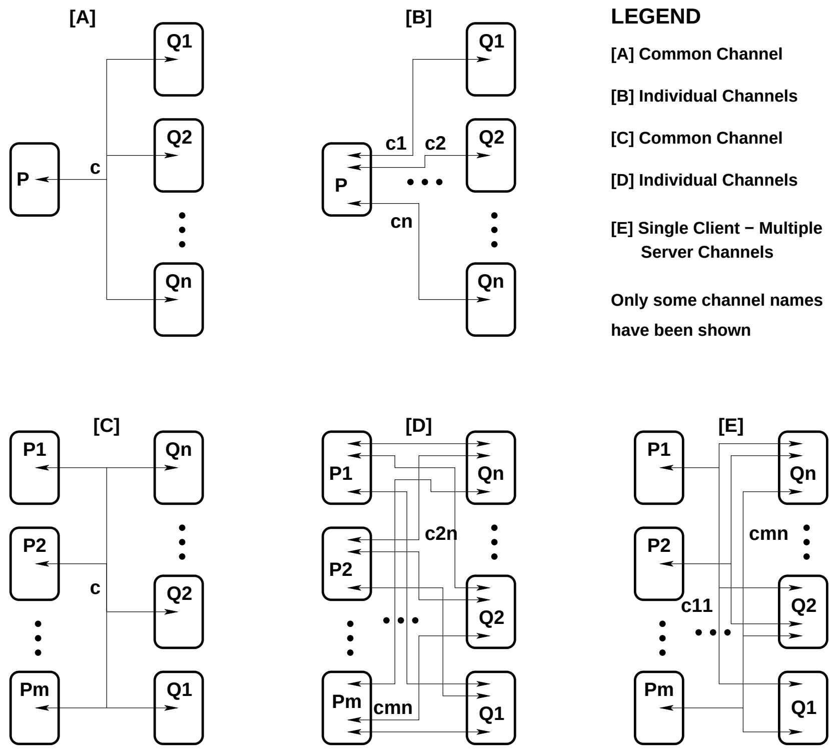


Figure 21.31: Some system channel configurations

We leave it as an exercise to provide schemas for each of the five cases ([A–E]) above

Concurrency Concepts — A Summary

Characterisation 21.100 • *Events* are atomic and instantaneous; they "occur".

⋆ Events are basic (primitive) elements of processes. Processes are, from a certain level of abstraction, composed from events.

⋆ Events are used to mark important points in the (temporal or partially ordered) history of a system (i.e., a process).

⋆ Typically events may stand for a process having reached a certain control (and data) state (a summary of past actions), or for some undefined (or undefinable) environment spontaneously wishing to interact, that is, synchronise and communicate with some process

. ■

Characterisation 21.101 • A *sequential process* is an ordered (i.e., sequential) set of operations (i.e., actions) on a data state. (Many processes will be cyclic.)

⋆ Some actions may simply change a data state. Others may cause the synchronisation between two processes and the communication, i.e., transfer of values from one process to the other.

. ■

Characterisation 21.102 *Blocked Process:*

- When a process is unable to progress, i.e., to commence a next action, then it is said to be *blocked.*
- A process description prescribes the conditions under which events may occur, and thus the conditions under which they may be blocked

. ■

Characterisation 21.103 • A *parallel process* is a usually unordered (i.e., not predictable) set of sequential process operations (i.e., actions) on own or shared data states

. ■

Characterisation 21.104 *Action:*

- Events usually "trigger" *actions*, that is, operations upon the data state of a process. As we shall later see, events may stand for output from one process and the corresponding input to another process, that is, for synchronisation and exchange of information between processes

. ■

Characterisation 21.105 *Channel:*

- Synchronisation of (e.g., two) and communication between processes "takes place" over (i.e., on) ***channels***. *Channels* allow processes connected to the channels to share events

. ■

Characterisation 21.106 *Behaviour:*

- Sets of observable sequences of events and/or actions of a process or a set of processes. Observations are usually made on “what goes on” on a channel (between processes). The sets may be finite or infinite

. ■

Characterisation 21.107 • A *trace* is a single sequence of events and/or actions of a postulated or actual process. A *trace* is either of finite or infinite length. A *behaviour* is a set of traces. A *process* usually denotes a behaviour

. ■

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Characterisation 21.108 *Environment:*

- Channels may be connected, at one end, to a process, but at the other end may be left "dangling". Such channels help define aspects of an *environment:* something "external" to or "outside" the collection of processes of main concern.
- Thus defined processes can share events with an environment: they can react to events from or "deliver" events to the environment

. ■

Topic 67
Communicating Sequential Processes, CSP

- In the previous lecture we have provided intuitive examples of concurrent specifications expressed in `RSL/CSP`.
  - ⋆ In those examples (of that notation) can be found a lot of syntactic details,
  - ⋆ that may clutter the presentation.
  - ⋆ In the present lecture we shall therefore show the `CSP` notation, a "purest" form of *Communicating Sequential Processes*,
  - ⋆ in order to show the utter elegance of the underlying concepts and their accompanying notation.

- This lecture thus goes back to the origins of `CSP`
  - ⋆ by presenting a "cleanest", simplest view of an essence of `CSP`.
  - ⋆ The "language" of `CSP` to be presented here is to processes what the λ-calculus is to functions.
  - ⋆ We shall only cover its language constructs and explain their meaning informally.
  - ⋆ We shall not delve into issues of mathematical models for the semantics of the `CSP` variant covered here.

- First we bring some preliminaries on processes and events.
- Then, in eleven "easy pieces", we cover the major process combinators ($\rightarrow$, $\Box$, $\sqcap$, ? [input], ! [output] and $\|$) as well as some basic and compound process expressions, and a few laws.

Preliminaries: Processes and Events

- By $\mathcal{P}, \mathcal{P}', \ldots, \mathcal{P}'', \mathcal{Q}, \ldots$ we mean processes. Not process descriptions, but processes "themselves".
- By Pn, Pn′, ..., Pn″, Qn, ... we mean process names (process names are process expressions).
- By Pe, Pe′, ..., Pe″, Qe, ... we mean process expressions, in general.

  Thus:

  Pn ≡ Pe

  gives the name Pn to the process expression (or description) Pe. Pn may occur (hence recursively) in Pe.
- By a, a', ..., a'', b, ... we mean events.

  Events are presently considered atomic. Later we shall structure events over (sets of) sets of values (and channels).

Process Combinators, Etcetera

stop: A Basic Process

- **stop:**

The process **stop** is unable to perform (issue, generate, participate in) any events.

Prefix

- $a \to P$

is a process which is **ready** to engage in the event a. If the event a occurs the process will then behave as P.

Definitions

- $Pn \equiv Pe$

Pn is an identifier (a name), and the expression, Pe, defines the process of that name to **behave** as the process expression Pe prescribes. That expression may contain the name Pn (as well as much else).

- *Example:*

$$\mathrm{Q} \equiv \mathrm{e} \rightarrow \mathrm{Q}$$

Q is the process whose behaviour is the singleton set of the infinite trace of the same event e.

$\Box$: External Nondeterministic Choice

- $P \Box Q$

Operationally you may think of any one trace of $P \Box Q$ being either P or Q. Which one is “selected” is nondeterministically determined by the environment of $P \Box Q$. The process $P \Box Q$ is available to engage in the events of either P or Q.

- *Example:*

$$\mathrm{P} \equiv \mathrm{requestA} \rightarrow \mathrm{performX} \;\Box\; \mathrm{requestB} \rightarrow \mathrm{performY}$$

P is the process which is willing to engage in either event *requestA* or *requestB*. If event *requestA* is chosen then P behaves like *performX*.

The **environment** offers the events *requestA* and *requestB*.

⊓: Internal Nondeterministic Choice

We write:

- $P \sqcap Q$

to denote the internal nondeterministic choice between processes P and Q. The *environment* has no influence over which of the two alternatives is chosen; but one is chosen “at random”.

- *Example:*

$$\mathrm{P} \equiv \mathrm{reqA} \to (\mathrm{actA1} \sqcap \mathrm{actA2}) \,\Box\, \mathrm{reqB} \to (\mathrm{actB1} \sqcap \mathrm{actB2})$$

Process P engages either in the behaviours *actA1* ⊓ *actA2* or *actB1* ⊓ *actB2* depending on the external nondeterministic (□) choices *reqA* and *reqB*. The process *actA1* ⊓ *actA2* behaves either like *actA1* or *actA2* — chosen nondeterministically by an internal choice. The situation is similar for process *actB1* ⊓ *actB2*.

CSP Law (I)

$$a \rightarrow (P \sqcap Q) \equiv (a \rightarrow P) \sqcap (a \rightarrow Q)$$

Compound Events

Sets of related events can be compounded. In CSP we can write:

$$\square\ e{:}\{a.1,a.2,a.3\} \cdot e \rightarrow P \quad \equiv \quad a.1 \rightarrow P \ \square\ a.2 \rightarrow P \ \square\ a.3 \rightarrow P7$$
$$\square\ i{:}\{1,2,3\} \cdot a.i \rightarrow P \ \equiv\ (a.1 \rightarrow P)\ \square\ (a.2 \rightarrow P)\ \square\ (a.3 \rightarrow P)$$

Input and Output

External choice, in `CSP`, corresponds to **in**put, internal choice to **out**put:

$$
\begin{aligned}
c\ ?\ k{:}K \to P(k) &\equiv \Box\ k{:}K \cdot c.k \to P(k)\\
d\ !\ k{:}K \to Q(k) &\equiv \Box\ k{:}K \cdot d.k \to P(k)
\end{aligned}
$$

In RSL **in**put and **out**put can be “mixed”:

channel
  $c{:}C,\ c'{:}C',\ c1{:}C1,\ c2{:}C2,\ \ldots,\ cn{:}Cn$
value
  /∗ either nondeterministic input ∗/
    let u = c1? **in** P(u) **end** ⌷ **let** v = c2? **in** Q(v) **end** ⌷ ...
  /∗ or nondeterministic output ∗/
    c1!e1 ; P′ ⌷ c2!e2 ; P″ ⌷ ...
  /∗ or both (mixed) ∗/
    c1!e1 ; P′ ⌷ **let** u = c? **in** P(u) **end** ⌷ c2!e2 ; P″ ⌷ ...

$\|$: Parallel Composition

- $P \parallel Q$

denotes the parallel composition of processes P and Q. Colloquially, i.e., speaking ‘operationally’, process $P \parallel Q$ describes a process as consisting of two other processes that “run in parallel” while cooperating on shared events.

Shared Events

Process expressions P and Q will often contain expressions listing the same, that is, "shared" events. Shared events are events of the same alphabetic name:

- $\alpha\ P$: alphabet of P, etc.

If:

$$\alpha\ \mathrm{P} \cap \alpha\ \mathrm{Q} = \{\mathrm{a,b,c}\}$$

then processes P and Q share events a, b, c and are thus willing to engage in these simultaneously.

$$\mathrm{x} \to \mathrm{P} \parallel \mathrm{x} \to \mathrm{Q} \ \equiv\ \mathrm{x} \to (\mathrm{P}\parallel\mathrm{Q})$$

If αP does not contain event z and if αQ does not contain event y then:

$$\mathrm{y} \to \mathrm{P} \parallel \mathrm{z} \to \mathrm{Q} \ \equiv\ (\mathrm{y} \to (\mathrm{P}\parallel(\mathrm{z}\to\mathrm{Q})))\ \square\ (\mathrm{z} \to ((\mathrm{y}\to\mathrm{P})\parallel\mathrm{Q}))$$

CSP Law (II)

if

$$P \equiv [\,]\ e{:}A \cdot e \rightarrow P(e)$$
$$Q \equiv [\,]\ e{:}B \cdot e \rightarrow Q(e)$$

then

$$\begin{array}{l} P \parallel Q \equiv \\ \quad [\,]\ e{:}A \setminus \alpha\, Q \cdot e \rightarrow (P(e) \| Q) \\ \quad [\,] \\ \quad [\,]\ e{:}B \setminus \alpha\, P \cdot e \rightarrow (P \| Q(e)) \\ \quad [\,] \\ \quad [\,]\ e{:}A \cap B \cdot e \rightarrow (P(e) \| Q(e)) \end{array}$$

end

Simple event	Events occur, have no time-duration, cause actions, change the control state	e
Input/-output	As events, but include actions: output, respectively input, latter changes the data state	c ! expr, c ? var
Process	P, Q are process expressions that designate sequences of one or more actions and events	P, Q, . . .
stop	The no-effect action	**stop**
Prefix	e→P is a process expression: event e followed by process P, where e may be c ! expr or c ? var. P is a process expression	e→P
External choice	P⌷Q is a process expression: P, Q are process expressions	P⌷Q
Internal choice	P⌈⌉Q is a process expression: P, Q are process expressions	P⌈⌉Q
Parallel composition	P, Q are process expressions. The designated processes proceed in parallel	P‖Q
Process definition	Pn is a process identifier, P a process expression; (a) an argument (a may be free in P), and where a process identifier in P is a process expression	Pn(a)≡P
Shared event	Upon event e the above process makes the transition to P‖Q, e is shared between P and Q.	(e→P) ‖(e→Q)

© Dines Bjørner
Fredsvej 11
DK-2840 Holte
Denmark
and © Springer
Tiergartenstraße 17
D 69121 Heidelberg
Germany

Topic 68

The RSL/CSP Process Combinators

- In an earlier lecture we formally introduced the concept of CSP-like processes.
- In an even earlier lecture we intuitively motivated and informally used a notation which is derived from CSP and has been adopted for RSL.
- In this lecture we will briefly, but systematically review this notation, the RSL/CSP "sublanguage", which is RSL-like.
- That is, this language is not exactly a subset of RSL. We have taken some liberties wrt. arrays of channels and how we name channels in function (i.e., process) type clauses.
- We shall elsewhere show that our deviation can be explained in terms of RSL.
- In the following we shall cover this RSL-like notation, syntactical construct by construct.

RSL-like Channels

- Channels are the means for synchronising processes and communicating values (i.e., messages).
- Channels
  - ⋆ lead from a "surrounding" outside (the environment) to defined processes,
  - ⋆ or lead to such a surrounding from defined processes,
  - ⋆ or channels are placed between, i.e., "infixed" defined processes,
  - ⋆ or combinations of the above.
- We may speak of single channels or of an indexed set of channels. The latter are intended wherever our system of processes involves similarly indexed sets of like processes.

- Channels must, in RSL, be declared:

```
type
   C /* C can designate any type */
   Cindex /* Cindex designates a finite set */
channel
   c1,c2,...,cn:C  /* n≥1 */
   { c[ i ]:C | I:CIndex }
```

- Channels ci can communicate values of type C.
- c is like an array of channels.
- c[i] for i ranging over the finite set of enumerations Cindex is otherwise like any channel ci.

RSL Communication Clauses

- Systems are either composed from a fixed set of processes or from a combination of one or more fixed and one or more sets of indexed processes.
- Correspondingly, we speak of fixed, constant named output/input communications, respectively of varying, indexed output/input communications.
- We presently treat the former kind of communications.

Simple Input/Output Clauses

- There are basically two communication clauses.
- First, input expressions:

  c ?
  let v = c ? **in** E(v) **end**

- The first clause above designates a value expression and expresses willingness to input a value from channel c.
- The second clause above also designates a value expression, with the embedded value expression c ? value being bound to variable v.

- The output clause:

```
c ! expr
```

- designates an output statement, that is, an expression, and expresses an offer of the value of expression expr for communication on channel c. As an expression it has the **Unit** type value ().

- Sometimes an input (`c ?`) is from an undefined process of a (globally) surrounding environment, and sometimes an output (`c ! expr`) is to an undefined process of a (globally) surrounding environment.
- And sometimes — in a set of processes — there are groups of (two or more) processes which define "matching" output/inputs — one or more of `c ! e` and one or more of `c ?`.
- We refer to examples already given in Figs. 21.169 and 21.171.

RSL Processes

Simple Process Definitions

```
value
  P: A → in c_i1,c_i2,...,c_im  /* m ≥ 0 */
         out c_o1,c_o2,...,c_on  /* n ≥ 0 */
         Unit
  P(a) ≡ $\mathcal{S}(a)$

  Q: Unit → in c_i1,c_i2,...,c_im  /* m ≥ 0 */
            out c_o1,c_o2,...,c_on  /* n ≥ 0 */
            Unit
  Q() ≡ $\mathcal{S}$(...)
```

- Process P takes [optional] input arguments in A,
- is willing to, i.e., may, receive input over channels c_i1, . . . , c_im, and
- is willing to, i.e., may, output over channels c_o1, . . . , c_om.
- Process P's signature ends with **Unit** to designate that no explicit value is returned, i.e., that the P process either recurses (i.e., "loops") indefinitely or "ends" with the interpretation of a clause of type **Unit.**
- Process Q takes no input and delivers no output, but is otherwise as is P.

The function, i.e., the process definition

value
  R: **Unit** $\rightarrow$ **any** B
  R() $\equiv$ ... ? ... ! ... b

designates a process which may engage in input/output over any channel, and which yields a value of type B.

Processes and Their Definitions

- Please note the distinction between a process definition or a process expression, on one hand, and a process, on the other hand.
- The former are pieces of text, syntactic “things”.
- The latter is a semantic phenomenon, invisible to the human eye!
- Processes communicate, not process expressions or process definitions. They prescribe communications.

Process Invocations

- Processes get started whenever a process invocation takes place.
- Invocations of processes are prescribed as follows:

  P(a), Q()

- The argument a can be thought of as a state.
- The process, as described by the named process definition, is started whenever a process invocation expression is elaborated.
- A recursive invocation, P(a′), then means that a state has been updated.

© Dines Bjørner
Fredsvej 11
DK-2840 Holte
Denmark
and © Springer
Tiergartenstraße 17
D 69121 Heidelberg
Germany

Example 21.172 *A Buffer Process Definition:*

```
type
  V
channel
  in_ch,out_ch:V
value
  Buffer: V* → in in_ch out out_ch Unit
  Buffer(q) ≡
    let v = in_ch? in Buffer(q⁀⟨v⟩) end
    □
    out_ch!hd q; Buffer(tl q)
```

■

Array Channel Process Definitions

- Let the intention be that CI_index and CJ_index designate finite, enumerable token sets.

```
type
  A_idx, B_idx
channel
  { c_in[ c ] | c:A_idx }, { c_out[ c ] | c:B_idx }
value
  P: a:A_idx × b:B_idx →
       in c_in[ a ] out c_out[ b ] Unit
  P(i,j) ≡
    ... let v = c_in[ i ] ? in
       ... c_out[ j ] ! e ... end ...
```

- The above process signature is nonstandard RSL.
- Note the binding of the channel array indices from the left of the → to the right.
- We will elsewhere show that the above is a shorthand for a more elaborate set of RSL **scheme** (and hence **class**) and **object** definitions and declarations.

```
value
  Q: Unit → in {c_in[c]|c:CA_index} out {c_out[c]|c:CB_index} Un
  Q() ≡
    ⌷ { let v = c_in[c′] ? in
        ⌈⌉ { c_out[c″] ! v | c″:CB_index } end | c′:CA_index }
```

Parallel Process Combinator

- Typically a system of concurrently operating components can be expressed as the parallel composition of component processes.
- Let P_i designate expressions.

```
P_1 || P_2 || ... || P_n
```

- The above expresses the parallel composition of n processes.
- Evaluation of each individual P_i in P_1||P_2||...||P_n proceeds in parallel.
- Earlier figures, including Figs. 21.169 and 21.171, illustrated systems (S1, S2, S3, S5, S5, sys and S6) of processes.

© Dines Bjørner Fredsvej 11 DK-2840 Holte Denmark and © Springer Tiergartenstraße 17 D 69121 Heidelberg Germany

Nondeterministic External Choice

- Let P_i designate expressions.
- Then:

```
P_1 ⌷ P_2 ⌷ ... ⌷ P_n
```

- expresses the parallel nondeterministic external choice between n processes.
- Let, for example (omitting type clauses),

```
P1() ≡ let v = c ? in E1(v) end
P2() ≡ let v = c ? in E2(v) end
Q() ≡ (c ! e)
R() ≡ (P1() ⌷ P2()) ∥ Q()
```

- The value of expression e is communicated to either the first or the second of the ▯ argument processes and hence is evaluated either under E1 or under E2.
- Which one is chosen (left, P1, or right P2), is not shown explicitly, but one is chosen.
- Wrt. (P1() ▯ P2()) we say that Q() is a surrounding process, and vice versa.
- (P1() ▯ P2()) is willing to engage in communication with its surrounding, and Q() likewise.

Nondeterministic Internal Choice

- Let P_i designate expressions all of which are of the same type.
- Then

```
P_1 ⌈⌉ P_2 ⌈⌉ ... ⌈⌉ P_n
```

- expresses the parallel nondeterministic internal choice between n processes.
- Either P_1, or P_2, or, . . . , or P_n is chosen — only the choice is internal nondeterministic, i.e., not dependent on any possibly surrounding processes.

Example 21.173 *A "Rolling a Dice" Process Definition:* To express the arbitrary selection among a finite set of enumerated possibilities we make use of nondeterministic internal choice

type
  Dice = one | two | three | four | five | six
value
  P: **Unit** → Dice
  $\mathrm{P}() \equiv \mathrm{one} \sqcap \mathrm{two} \sqcap \mathrm{three} \sqcap \mathrm{four} \sqcap \mathrm{five} \sqcap \mathrm{six}$

Invocation of P() "randomly" yields a face of a dice. ■

Interlock Combinator

- Sometimes it is necessary to force two concurrent processes to prioritise their mutual communication — over other such.
- For that RSL offers the *interlock combinator:*

pe_1 $\#$ pe_2.

- The above interlock composition is evaluated as follows:
  - ⋆ The two expressions are evaluated concurrently.
  - ⋆ If one of them comes to an end before the other,
  - ⋆ evaluation continues with that other.
  - ⋆ However, during the concurrent evaluation,
  - ⋆ any communication external to pe_1 $\#$ pe_2 is prevented.
  - ⋆ Thus pe_1 $\#$ pe_2 expresses that the two processes
  - ⋆ are forced to communicate only with one another,
  - ⋆ until one of them terminates.

Summary

We provide a check list summary of RSL/CSP clauses:

- *Channel:* **channel** c:C
- *Input:* c ? and **let** v = c ? **in** Pe **end**
- *Output:* c ! r
- *Process expressions:* Pe_1 ; Pe_2 ; ... ; Pe_n
- *Parallelism:* Pe_1 ∥ Pe_2 ∥ ... ∥ Pe_n
- *External nondeterminism:* Pe_1 ⌷ Pe_2 ⌷ ... ⌷ Pe_n
- *Internal nondeterminism:* Pe_1 ⌈⌉ Pe_2 ⌈⌉ ... ⌈⌉ Pe_n
- *Interlocking:* Pe_1 ╫ Pe_2
- *Process definition:* Pn: A → **in** c_i **out** c_j **Unit**
  Pn(a) ≡ Pe

A Note of Caution

- We remind the student that the present lectures notes' function signatures,
  - ⋆ when it comes to such functions which define processes using channels (etc.),
  - ⋆ go beyond "standard" (i.e., tool-supported) RSL,
  - ⋆ in allowing a kind of "dependent" types:

```
type
  X_Idx, Y_Idx, M, A, ...
channel
  { c[x,y]:M | x:X_Idx,y:Y_Idx }
value
  f: x:X_Idx × A → in { c[x,y] | y:Y_Idx } ...
```

  - ⋆ In the function f signature x is being bound "to the left" of →
  - ⋆ and is being "used" "to the right" of →,
  - ⋆ in delimiting the channels from which to input.

Topic 69

Translation Schemas

Stage I: An Applicative Schema

- Let us consider the following schema:

type
  A, B
value
  f: A → **Unit**
  g: A → A
  h: A → **Unit**

  f(a) ≡ (**let** a′ = g(a) **in** f(a′) **end** ⌈⌉ h(a) ; f(a))

© Dines Bjørner Fredsvej 11 DK-2840 Holte Denmark and © Springer Tiergartenstraße 17 D 69121 Heidelberg Germany

Stage II: A Simple Reformulation

- The above schema defines the behaviour of f as a nondeterministic internal choice behaviour between two processes: **let** $a' = g(a)$ **in** $f(a')$ **end** and $h(a)$; $f(a)$. Let us call them g' and h':

type
  A, B
value
  f: $A \rightarrow$ **Unit**
  g: $A \rightarrow A$
  g′,h,h′: $A \rightarrow$ **Unit**

  $f(a) \equiv g'(a) \sqcap h'(a)$
  $g'(a) \equiv$ **let** $a' = g(a)$ **in** $f(a')$ **end**
  $h'(a) \equiv h(a)\ ;\ f(a)$

- Let us examine the above:
  - ⋆ It seems that f is a main process.
  - ⋆ It seems that a is like a state variable, being used and updated (by g'), or just used and "passed on", by h'.
  - ⋆ In other words, the two processes G and H both require access to the shared state,
  - ⋆ but the two processes' g, respectively h, "actions" cannot proceed in parallel.
- Observe that f is not recursive, but g' and h' are.

Stage III: Introducing Parallelism

- What about the following idea:
  - ⋆ "Split out", i.e., decompose, f into two three parallel processes F, G and H.
  - ⋆ In this case F "maintains" the global state a,
  - ⋆ and G and H reread, respectively rewrite that global state:

```
type
  A, B
channel
  fg:A, fh:A
value
  S: A → Unit
  F: A → out fg,fh  Unit
  G: Unit → in fg  Unit
  H: Unit → in fh  Unit
  g: A → A
  h: A → Unit

  S(a) ≡ F(a) ∥ G() ∥ H()
  F(a) ≡  fg!a ⌈⌉ fh!a
  G() ≡ let a = fg ? in let a′ = g(a) in F(a′) end end
  H() ≡ let a = fh ? in h(a) ; F(a) end
```

Stage IV: A Simple Reformulation

- Instead of using the "tail" recursive invocations of F from both G and H, "passing" on appropriate arguments to the F process,
- we communicate, over a (new) channel, a possibly updated value of the argument (a').
- Since H need not communicate any new A value, we let it, for sake of "symmetry", communicate a "tick", indicating, for whatever it is worth, completion.
- A variant of F, G and H could thus be:

```
type
  Tick == tick
channel
  fg:A, fh:A, gf:A, hf:Tick
value
  F: A → out fg,fh in gf,hf  Unit
  G: Unit → in fg out gf  Unit
  H: Unit → in fh out hf  Unit

  F(a) ≡
    (fg!a ; let a′ = gf ? in F(a′) end)
    ⌈⌉
    (fh!a ; let t = hf ? in F(a) end)

  G() ≡ let a = fg ? in let a′ = g(a) in gf!a′  end end ; G()
  H() ≡ let a = fh ? in gh!tick ; h(a) end ; H()
```

Stage Relations

- Now we have to stop and consider!
- Is the above "development",
  - ⋆ from **stage I**
  - ⋆ via **stage II**
  - ⋆ and **stage III**
  - ⋆ to **stage IV**,

  correct?
- And, what, after all, do we mean by correctness?

- It is clear that the four stages of formulas do not exhibit the same functions, and, where they do "share" some function names, that the function signatures are not the same.
- So, from that point of view, the four stages do not "compute the same thing".
- But they are comparable.
- We claim that the sequence of state updates of the four models are the same!

Stage V: An Imperative Reformulation

- In an earlier lecture we saw that an applicative function could be "imperialised".
- So we now do with F:

```
variable
  σ:A
value
  F: Unit → read,write σ  out fg,fh in gf,hf  Unit
  F() ≡
    (fg!σ ; σ := gf ? ; F())
    ⌈⌉
    (fh!σ ; let t = hf ? in F() end)
```

We can even turn tail recursion into an imperative loop:

```
variable
  $\sigma$:A
value
  F: Unit → read,write $\sigma$ out fg,fh in gf,hf Unit
  F() ≡
    while true do
      (fg!$\sigma$ ; $\sigma$ := gf ? )
      $\prod$
      (fh!$\sigma$ ; let t = hf ? in skip end)
    end
```

Some Remarks

- This ends our informal, yet systematic, sequence of stages of "comparable" and "believably correct" developments.
- The idea of this part of the lecture has been to give you some hints as how to turn applicative and recursive function definitions into imperative process definitions.

Topic 70

Parallelism and Concurrency: A Discussion

CSP and RSL/CSP

- This set of lectures, on parallel specification programming, has focused on CSP,
- and for good reasons.
- CSP provides an elegant way of expressing concurrency.
- Furthermore, CSP blends well with RSL.
- Learning RSL/CSP will enable the course student to quickly adapt to, i.e., learn and use, “pure” CSP.
- “Pure” CSP — with its tool support for model checking, a means of proving that certain CSP satisfy expected properties — is very useful as a separate tool for investigating specific specifications of (specific) concurrent system proposals.