13

Message and Live Sequence Charts

Christian Krog Madsen is chief author of this chapter [316, 317].

- The **prerequisite** for studying this chapter is that you have an all-round awareness of abstract specification (principles and techniques).
- The **aims** are to introduce the concepts of message sequence charts and of live sequence charts, and to relate these sequence charts to RSL/CSP.
- The **objective** is to enable the reader to expand the kind of phenomena and concepts that can be formally modelled using message sequence charts and live sequence charts — or, we suggest, live sequence charts in conjunction with, for example, RSL.
- The **treatment** ranges from systematic to formal.

Live sequence charts (LSC) is a graphical language introduced by Damm and Harel [89] for specifying interactions between components in a system. It is an extension of the language of message sequence charts (MSC). MSCs are frequently used in the specification of telecommunication systems and are closely related to the sequence diagrams of UML [59, 237, 382, 440]. Both the graphical and textual syntax of MSCs are standardised by the ITU in Recommendation Z.120 [227–229]. The standard gives an algebraic semantics of MSCs. LSC extends MSC by promoting conditions to first-class elements and providing notations for specifying mandatory and optional behaviour.

Reader's Guide

The description material on basic (and on high-level) MSCs in Sects. 13.1.2–13.1.3 and on LSC in Sect. 13.2.1 is intended as quick tutorials as well as for quick reference. Sect. 13.3, on the important computer science topic of *process algebra*, and Sect. 13.4, on an algebraic semantics of LSCs, are both rather involved and may seem a bit detached from the context. The reader is encouraged to refer to the example in Sect. 13.2.2 for an understanding of LSCs, and to its continuation in Sect. 13.4.3 to see how the algebraic semantics of a chart is derived using the material of Sect. 13.3.

13.1 Message Sequence Charts

13.1.1 The Issues

In this section we describe message sequence charts (MSCs). They are a graphical notation for specifying sequences of messages exchanged between behaviours.[1] We describe the components of MSCs and then provide a formalisation of the syntax in RSL. We follow the syntax requirements defined by Reniers [422, 423]. Finally, we give a trace semantics of MSCs.

Message sequence charts were first standardised by the CCITT (now ITU-T) as Recommendation Z.120 in 1992 [227]. The standard was later revised and extended in 1996 [228] and in 1999 [229]. The original standard specified the components of an MSC. The 1996 standard also specified how several MSCs (called *basic* MSCs) can be combined to form an MSC document, in which the relation between the basic MSCs is defined by a *high-level* MSC (HMSC). The most recent standard provides additional facilities for specifying the data that is passed in messages and also allows in-line expressions.

13.1.2 Basic MSCs (BMSCs)

Informal Presentation

A basic MSC (BMSC) consists of a collection of instances. An instance is an abstract entity on which events can be specified. Events are message inputs, message outputs, actions, conditions, timers, process control events and coregions. An instance is denoted by a hollow box with a vertical line extending from the bottom. The vertical line represents a time axis, where time runs from top to bottom. Each instance thus has its own time axis, and time may progress differently on two axes. Events specified on an instance are totally ordered in time. Events execute instantaneously and two events cannot take place at the same time. Events on different instances are partially ordered, since the only requirement is that message input by one instance must be preceded by the corresponding message output in another instance.

Actions are events that are local to an instance. Actions are represented by a box on the timeline with an action label inside. Actions are used to specify some computation that changes the internal state of the instance.

A *message output* represents the sending of a message to another instance or the environment.

A *message input* represents the reception of a message from another instance or the environment. For each message output to another instance there must be a matching message input.

[1] An alternative way of phrasing *sequences of messages exchanged between behaviours* is *events shared between two behaviours where these events may involve the communication of information.*

A *message exchange* consists of a message output and a message input. A message exchange is represented as an arrow from the timeline of the sending instance to the timeline of the receiving instance. In case of messages exchanged with the environment, the side of the diagram can be considered to be the timeline of the environment. Each arrow is labelled with a message identifier. Message exchange is asynchronous, i.e., message input is not necessarily simultaneous with message output.

Example 13.1 Figure 13.1 shows an MSC with two instances, A and B. Instance A sends the message m_1 to instance B followed by message m_2 sent to the environment. B then performs some action, a, and sends the message m_3 to A. ▪

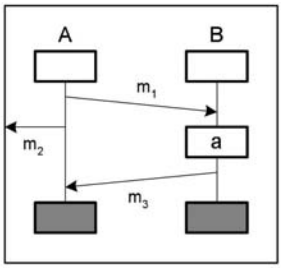

Fig. 13.1. Message and action events

Example 13.2 Figure 13.2 shows two situations that violate the partial order induced by message exchange. Thus it is an invalid MSC. Because events are totally ordered on an instance timeline, the reception of message m_1 precedes the sending of m_1. This conflicts with the requirement that message input be preceded by message output.

The exchange of messages m_2 and m_3 illustrates another situation that violates the partial order, as shown by the following informal argument. Let the partial order be denoted $\leq$ and let the input and output of message m be denoted by $in(m)$ and $out(m)$, respectively. Using the total ordering on events on an instance timeline we have:

$$in(m_3) \leq out(m_2)$$
$$in(m_2) \leq out(m_3)$$

Using the partial ordering on message events we have

$$out(m_2) \leq in(m_2)$$

Now, by transitivity of $\leq$, $in(m_3) \leq out(m_3)$, thus violating the partial ordering on message events. ▪

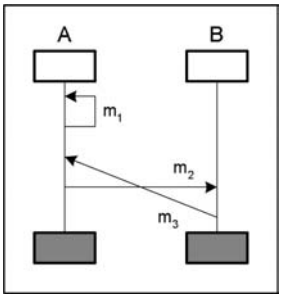

Fig. 13.2. Illegal message exchanges

Conditions describe a state that is common to a subset of instances in an MSC. Conditions in MSCs have no semantic importance and merely serve as documentation. (As we shall later see, they do have meaning in LSCs.) Conditions are represented as hexagons extending across the timelines of the instances for which the condition applies. The condition text is placed inside the hexagon.

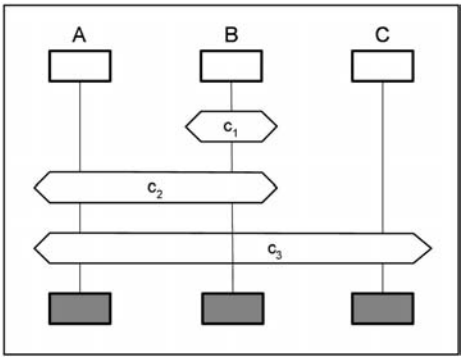

Fig. 13.3. Conditions

Example 13.3 Figure 13.3 illustrates conditions. Condition c_1 is local to instance B. Condition c_2 is a shared condition on instances A and B. Condition c_3 is a shared condition on instances A and C. Note that the timeline of B is passed through the hexagon for condition c_3 to indicate that B does not share condition c_3. ▪

There are three *timer* events: *timer set*, *timer reset* and *timeout*. Timers are local to an instance. The setting of a timer is represented by an hourglass symbol placed next to the instance timeline and labelled with a timer identifier. Timer reset is represented by a cross ($\times$) linked by a horizontal line to the timeline. Timer timeout is represented by an arrow from the hourglass symbol to the timeline. Every timer reset and timeout event must be preceded

by the corresponding timer set event. There is no notion of quantitative time in MSC, so timer events are purely symbolic. Extensions of MSC with time have been studied in [38, 280, 296].

Example 13.4 Figure 13.4 shows the syntax for timer events. On instance A, the timer T is set and subsequently timeout occurs. On instance B, the timer T' is set and subsequently reset. ▪

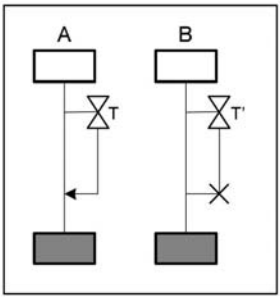

Fig. 13.4. Timer events

An instance may create a new instance, which is called *process creation.* An instance may also cause itself to terminate. This is called *process termination.* Process creation is represented by a dashed arrow from the timeline of the creating instance to a new instance symbol with associated timeline. Process termination is represented by a cross as the last symbol on the timeline of the instance that terminates.

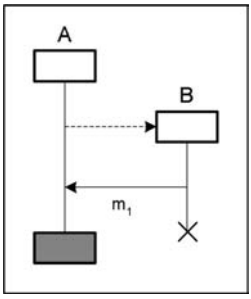

Fig. 13.5. Process creation and termination

Example 13.5 Figure 13.5 shows the creation of instance B by instance A and the subsequent termination of B. ▪

Coregions are parts of the timeline of an instance where the usual requirement of total ordering is lifted. Coregions are represented by replacing part of the fully drawn timeline with a dashed line. Within a coregion only message

exchange events may be specified and these events may happen in any order, regardless of the sequence in which they are specified. Message exchanges between two instances may be ordered in one instance and unordered in the other instance.

Example 13.6 Figure 13.6 illustrates a coregion in instance B. Because of the coregion, there is no ordering on the input of messages m_1 and m_2 in instance B, so they may occur in any order. ▪

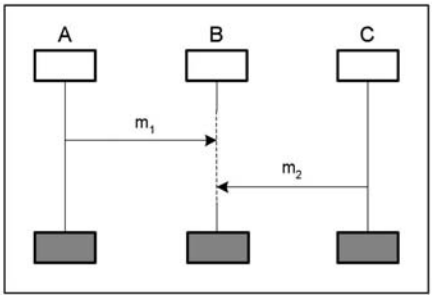

Fig. 13.6. Coregion

In order to increase the readability of complex MSCs, the standard specifies a form of hierarchical decomposition of complex diagrams into a collection of simpler diagrams. This is known as *instance decomposition*. For each decomposed instance there is a sub-MSC, which is itself an MSC. The single instance that is decomposed is represented by more than one instance in the sub-MSC. The behaviour observable by the environment of the sub-MSC should be equivalent to the observable behaviour of the decomposed instance.

Example 13.7 In Fig. 13.7 instance B is decomposed into two instances, B_1 and B_2 in the sub-MSC. The message events in which B participates are represented as message exchanges with the environment in the sub-MSC. The message m_{int} exchanged between B_1 and B_2 is internal to the decomposed instance, and is thus not visible in the main MSC. ▪

An Example BMSC

Example 13.8 *A Basic Message Sequence Chart:* Figure 13.8 shows an example BMSC that displays most of the event types discussed above. The chart contains three instances, A, B and C. Five events are specified on instance A: message output of a message labelled *Msg1* to instance B, a local action *Act1*, a condition *Cond1* shared with B, message output of *Msg4* and message input of *Msg5*. Seven events are specified on instance B: input of message *Msg1* from A, a process creation event creating instance C, two message exchanges with C, a condition shared with A, and a coregion with two

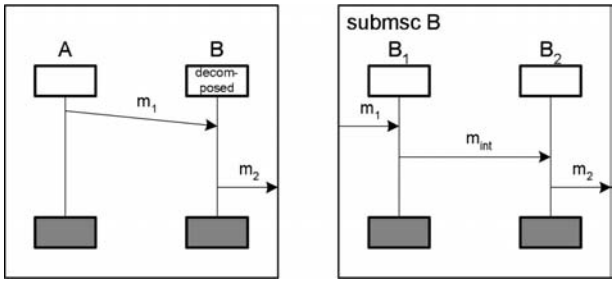

Fig. 13.7. Instance decomposition

message exchanges with A. Note that B may either receive *Msg4* and then send *Msg5*, or may send *Msg5* and then receive *Msg4*. Instance C has six events: its creation by B, the setting of a timer, two message exchanges with B, timer timeout and subsequent process termination. ▪

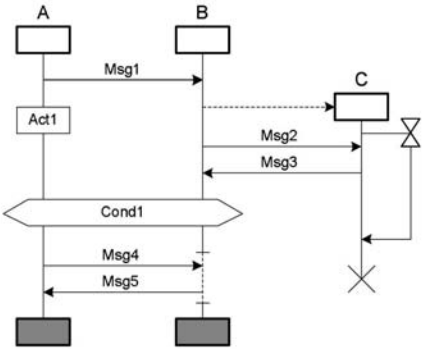

Fig. 13.8. A basic message chart example

An RSL Model of BMSC Syntax

We first formalise basic message sequence charts. We defer the discussion of well-formedness conditions to Section 13.1.6.

Definition. By a *basic message sequence chart* we shall understand a structure as formalised in this section and in Sect. 13.1.8. ▪

.

```
scheme BasicMessageSequenceChart =
  class
    type
      BMSC′ = BMSC_Name×InstanceSpec*×Body*,
      InstanceSpec = Inst_Name×Kind,
      Kind = Type×Kind_Name,
      Type == System|Block|Process|Service|None,
      Body = Instance|Note,
      Instance == mk_Inst(instn:Inst_Name,kind:Kind,evtl:Event*),
      Note == mk_Note(t:Text),
      Event =
        ActionEvent|MessageEvent|ConditionEvent|TimerEvent|
        ProcessEvent|CoregionEvent,
      ActionEvent == mk_Action(actname:Act_Name),
      MessageEvent ==
        mk_Input(inpid:MsgID,inpar:Par_Name*,inaddr:Address)|
        mk_Output(outid:MsgID,outpar:Par_Name*,outaddr:Address),
      ConditionEvent == mk_Condition(conname:Con_Name,share:Share),
      TimerEvent ==
        mk_Set(setname:TimerId,dur:Duration)|
        mk_Reset(resetname:TimerId)|
        mk_Timeout(toname:TimerId),
      ProcessEvent == mk_Create(name:Inst_Name,par:Par_Name*)|mk_Stop,
      CoregionEvent == mk_Concurrent(mess:MessageEvent*),
      MsgID ==
        mk_MsgN(mn:Msg_Name,parn:Par_Name*)|
        mk_MsgID(mid:Msg_Name,min:MsgInst_Name,parid:Par_Name*),
      Address == mk_Env|mk_InstName(name:Inst_Name),
      Share == mk_None|mk_All|mk_Shared(instl:Inst_Name*),
      TimerId ==
        mk_Tn(nametn:Timer_Name)|
        mk_Tid(nametid:Timer_Name,tin:TimerInst_Name),
      Duration == mk_None|mk_Name(name:Dur_Name),
      BMSC_Name,
      Inst_Name,
      Kind_Name,
      Act_Name,
      Par_Name,
      Con_Name,
      Timer_Name,
      TimerInst_Name,
      Dur_Name,
      Msg_Name,
      MsgInst_Name
  end
```

Annotations

- A BMSC has a name, a sequence of instance specifications and a sequence of body elements.
- An instance specification has an instance name and an instance kind.
- An instance kind has a type and a name.
- The type of an instance is either missing or is one of system, block, process or service.
- A body element is either an instance or a note.
- An instance has an instance name, an instance kind and a sequence of events.
- A note is a textual description or comment.
- An event is an action, message, condition, timer, process or is a coregion event.
- An action event has a name.
- A message event is either a message input or a message output. A message input is characterised by a message identifier, a possibly empty sequence of input parameters and an address identifying the sender. A message output has a message identifier, a possibly empty sequence of output parameters and an address identifying the recipient.
- A condition event has a name and an identification of the instances that share the condition.
- A timer event is the setting of a timer, the resetting of a timer or a timeout. All are characterised by a timer identifier, and, additionally, timer setting may specify a duration.
- A process event is either a process creation or a process termination. A process creation gives a name and a sequence of parameters to the new process.
- A coregion event contains a sequence of message events.
- An address is either the environment or the name of an instance.
- A condition may be local to an instance shared by all instances or shared by a subset of instances.
- A timer identifier is either a timer name, or a timer name and a timer instance name.
- A (timer-specified) duration is either unspecified or has a name.
- Names are further unspecified entities. ∎

13.1.3 High-Level MSCs (HMSCs)

An Informal Presentation

We now extend the above definition of BMSCs to allow several BMSCs to form an MSC document. To provide the link between BMSCs the high-level message sequence chart (HMSC) is defined.

A HMSC consists of a number of nodes, each representing a BMSC, connected with arrows. One node is the start node and several nodes may be

end nodes. Arrows denote vertical composition of the BMSCs they connect, i.e., the events of the origin BMSC occur first, followed by the events of the destination BMSC. Nodes may have arrows to several other nodes, indicating alternatives. In that case the origin BMSC is composed vertically with one of the alternative destination BMSCs. The graph of nodes and arrows may have loops, indicating iteration.

Nodes are represented by circles or rounded rectangles labelled with the name of the BMSC it denotes. Start nodes are indicated by an upside-down triangle (∇) with an arrow pointing to the node. End nodes are indicated by a triangle (Δ) pointed to by an arrow from the node. *Connectors* may be introduced to improve legibility. When connectors are used, each node may have at most one incoming arrow and one outgoing arrow. Connectors then serve as junctions for arrows, where one incoming arrow may split into several outgoing arrows or vice versa. Connectors are represented as small circles. The annotations of the formal model of the syntax of HMSCs provide more specific details, see below.

An Example HMSC

Example 13.9 *A High-Level Message Chart:* Figures 13.9–13.10 show a simple HMSC with three BMSCs. The chart models a client-server system, where a server offers some service, which the client can access. The start node of the HMSC is the BMSC *Init* in which the client logs on to the server and the server responds with a confirmation. Then one or more cycles of the BMSC *Transfer* follow, in which the client requests a resource and the server responds by returning that resource. Finally, the client logs off and the server closes the connection. ▪

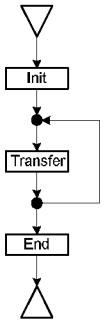

Fig. 13.9. HMSC example, part 1 of 2

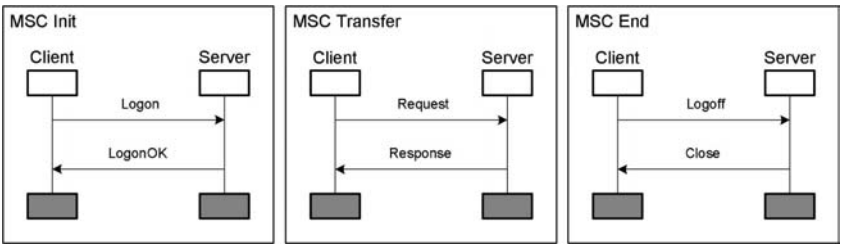

Fig. 13.10. HMSC example, part 2 of 2

13.1.4 An RSL Model of HMSC Syntax

Definition. By a *high-level message sequence chart* we shall understand a structure as formalised in this section, in Sects. 13.1.6 and 13.1.8 and as a solution to Exercise 13.3. ▪

The formalisation of HMSCs is simple, given the formalisation of BMSCs.

context: BasicMessageSequenceChart

scheme HighLevelMessageSequenceChart =
  extend BasicMessageSequenceChart **with**
  class
    type
      HMSC′ = (BMSC_Name $\overrightarrow{m}$ BMSC)
        × (BMSC_Name $\overrightarrow{m}$ BMSC_Name-**set**)
        × BMSC_Name
        × BMSC_Name-**set**
  end

Annotations

- A high-level message sequence chart is composed of a mapping of BMSC names to BMSCs,
- a set of outgoing arrows for each BMSC,
- a start node
- and a possibly empty set of end nodes. ▪

13.1.5 MSCs Are HMSCs

Definition. By a *message sequence chart* we shall understand a high-level message sequence chart. ▪

13.1.6 Syntactic Well-formedness of MSCs

Now that we have defined the full syntax of MSCs we are ready to specify the requirements for a chart to be well-formed. First, we specify conditions for a BMSC to be well-formed. These conditions were derived by Reniers [423].

```
context: HighLevelMessageSequenceChart
scheme WellformedBMSC =
extend HighLevelMessageSequenceChart with
class
  type BMSC = {| b:BMSC′•wf_BMSC(b) |}

  value
   wf_BMSC:BMSC′ → Bool
   wf_BMSC(n,s,b) ≡
     let
      inst=instances(n,s,b),
      instnames={instn(i)|i:Instance•i ∈ elems inst}
     in
      /* 1 */
      (∀ j,k:Nat•
           j ≠ k∧{j,k}⊆inds inst⇒
              inst(j) ≠ inst(k)∧instn(inst(j)) ≠ instn(inst(k)))∧
      /* 2 */
      (s ≠ ⟨⟩⇒
           (∀ i:Instance•
              (i ∈ elems inst) ≡ ((instn(i),kind(i)) ∈ elems s)))∧
      /* 3 */
      ({name(a)|
         a:Address•
           ∃ i:Instance,inpid:MsgID,pl:Par_Name*•
              i ∈ elems inst∧a ≠ mk_Env∧
              mk_Input(inpid,pl,a) ∈ elems inputEvts(i)} ∪
        {name(a)|
         a:Address•
           ∃ i:Instance,inpid:MsgID,pl:Par_Name*•
              i ∈ elems inst∧a ≠ mk_Env∧
              mk_Input(inpid,pl,a) ∈ elems outputEvts(i)}⊆instnames)∧
      /* 4 */
      (∀ i:Instance•
           i ∈ elems inst⇒
              (∀ evt,evt′:MessageEvent•
                 (evt ∈ inputEvts(i)∧evt′ ∈ inputEvts(i)∧
                    inpid(evt)=inpid(evt′)∧inaddr(evt)=inaddr(evt′)⇒
                       evt=evt′)∧
                 (evt ∈ outputEvts(i)∧evt′ ∈ outputEvts(i)∧
                    outid(evt)=outid(evt′)∧outaddr(evt)=outaddr(evt′)⇒
                    evt=evt′)))∧
      /* 5 */
```

```
(∀ i:Instance•
    i ∈ elems inst⇒
      (∀ mi:MsgID,pl:Par_Name*,inaddr:Address•
          mk_Input(mi,pl,inaddr) ∈ inputEvts(i)∧inaddr ≠ mk_Env⇒
            mk_Output(mi,pl,mk_InstName(instn(i))) ∈
              outputEvts(lookup(name(inaddr),b)))∧
      (∀ mi:MsgID,pl:Par_Name*,outaddr:Address•
          mk_Output(mi,pl,outaddr) ∈ outputEvts(i)∧
          outaddr ≠ mk_Env⇒
            mk_Input(mi,pl,mk_InstName(instn(i))) ∈
              inputEvts(lookup(name(outaddr),b))))∧
/* 6 */
~is_cyclic(
     {ss|
       ss:S×S,sss:(S×S)-set•
         ss ∈ sss∧
         sss ∈
           {po_inst(i,el,{}) ∪ po_comm(i,el)|
           i:Inst_Name,k:Kind,el:Event*•
             mk_Inst(i,k,el) ∈ inst}})∧
/* 7 */
(∀ i:Instance•
    i ∈ elems inst⇒
      (∀ c:ConditionEvent•
          c ∈ evtl(i)⇒
            case share(c) of
              mk_Shared(il) →
                (∀ i:Inst_Name•
                    i ∈ il⇒
                      (∃ k:Kind,el:Event*•
                        mk_Inst(i,k,el) ∈ b)),
              _ → true
            end))∧
/* 8 */
(∀ i:Instance•
    i ∈ inst⇒
      (∀ cn:Con_Name,sh:Share•
          mk_Condition(cn,sh) ∈ evtl(i)⇒
            case sh of
              mk_None → true,
              mk_All →
                (∀ i′:Instance•
                    i′ ∈ inst⇒
                      len ⟨c|c in evtl(i)•c=mk_Condition(cn,sh)⟩=
                        len ⟨c|c in evtl(i′)•
                            c=mk_Condition(cn,mk_All)⟩),
              mk_Shared(il) →
                (∀ i′:Instance•
                    i′ ∈ inst∧instn(i′) ∈ elems il⇒
```

```
                                        len ⟨c|c in evtl(i)•c=mk_Condition(cn,sh)⟩=
                                          len ⟨c|c in evtl(i')•
                                              ∃ il':Inst_Name*•
                                              c=mk_Condition(cn,mk_Shared(il'))∧
                                                elems il'=
                                                  (elems il \ {instn(i')}) ∪
                                                    {instn(i)}⟩)
                        end))∧
    /* 9 */
    (∀ i:Instance•
          i ∈ inst⇒
              (∀ n:Inst_Name,p:Par_Name*•
                  mk_Create(n,p) ∈ evtl(i)⇒
                      n ∈ instnames∧n ≠ instn(i)))∧
    /* 10 */
    (let
          pcl=
              ⟨⟨name(Event_to_ProcessEvent(pc))|
                      pc in evtl(Body_to_Instance(i))•
                          ∃ n:Inst_Name,p:Par_Name*•
                              pc=mk_Create(n,p)⟩|i in b•i ∈ inst⟩
      in
          (∀ l:Inst_Name*•l ∈ elems pcl⇒len l=card elems l)∧
          (∀ j,j':Nat•
              {j,j'}⊆inds pcl∧j ≠ j'⇒
                  elems pcl(j) ∩ elems pcl(j')={})
      end)
end,

instances:BMSC → Instance*
instances(n,s,b) ≡
    ⟨Body_to_Instance(i)|i in b•(∀ t:Text•i ≠ mk_Note(t))⟩,

inputEvts:Instance → MessageEvent*
inputEvts(i) ≡
    ⟨Event_to_MessageEvent(e)|
        e in evtl(i)•
            (∃ inpid:MsgID,inpar:Par_Name*,inaddr:Address•
                e=mk_Input(inpid,inpar,inaddr))⟩,

outputEvts:Instance → MessageEvent*
outputEvts(i) ≡
    ⟨Event_to_MessageEvent(e)|
        e in evtl(i)•
            (∃ outid:MsgID,outpar:Par_Name*,outaddr:Address•
                e=mk_Input(outid,outpar,outaddr))⟩,

lookup:Inst_Name×Body* ~→ Instance
lookup(i,bl) ≡
```

```
      case hd bl of
        mk_Inst(i′,_,_) →
          if i=i′ then Body_to_Instance(hd bl) else lookup(i,tl bl) end,
          → lookup(i,tl bl)
      end
    pre (∃ k:Kind,el:Event*•mk_Inst(i,k,el) ∈ bl)

type
    Dir == In|Out,S=Dir×(Inst_Name×Inst_Name×MsgID)

value
    po_inst:Inst_Name×Event*×S-set → (S×S)-set
    po_inst(i,el,prev) ≡
      if el=⟨⟩ then {}
      else
        case hd el of
          mk_Input(mi,p,ia) →
            {(n,(In,(i,name(ia),mi)))|n:S•n ∈ prev} ∪
            po_inst(i,tl el,{(In,(i,name(ia),mi))}),
          mk_Output(mi,p,oa) →
            {(n,(Out,(i,name(oa),mi)))|n:S•n ∈ prev} ∪
            po_inst(i,tl el,{(Out,(i,name(oa),mi))}),
          mk_Concurrent(mel) →
            {(n,(In,(i,ia,mi)))|
              n:S,ia:Inst_Name,mi:MsgID,p:Par_Name*•
                n ∈ prev∧mk_Input(mi,p,mk_InstName(ia)) ∈ mel} ∪
            {(n,(Out,(i,oa,mi)))|
              n:S,oa:Inst_Name,mi:MsgID,p:Par_Name*•
                n ∈ prev∧mk_Output(mi,p,mk_InstName(oa)) ∈ mel} ∪
            po_inst(
                  i,tl el,
                  {(In,(i,ia,mi))|
                   ia:Inst_Name,mi:MsgID,p:Par_Name*•
                     mk_Input(mi,p,mk_InstName(ia)) ∈ mel} ∪
                  {(Out,(i,oa,mi))|
                   oa:Inst_Name,mi:MsgID,p:Par_Name*•
                     mk_Output(mi,p,mk_InstName(oa)) ∈ mel}),
          _ → po_inst(i,tl el,prev)
        end
      end,

    po_comm:Inst_Name×Event* → (S×S)-set
    po_comm(i,el) ≡
      if el=⟨⟩ then {}
      else
        case hd el of
          mk_Output(mi,p,oa) →
            {((Out,(i,name(oa),mi)),(In,(name(oa),i,mi)))} ∪
            po_comm(i,tl el),
```

```
                _ → po_comm(i,tl el)
            end
        end,

    is_cyclic:(S×S)-set → Bool
    is_cyclic(sss) ≡
        (∃ s:S* •
            (∀ i:Nat•i>0∧i < len s⇒(s(i),s(i+1)) ∈ sss)∧
            s(1)=s(len s))
end
```

Annotations

- A BMSC is well-formed if each of the following conditions hold:
  1. In a BMSC instances are uniquely named.
  2. If an interface is specified for a BMSC, then for each instance in the interface there must be an instance with the same name and kind in the body of the chart and vice versa.
  3. Every input and output event must reference instances which are declared in the body of the chart.
  4. On an instance there may be at most one message input with a given message identifier and address. On an instance there may be at most one message output with a given message identifier and address.
  5. For each message output to an instance, there must be a corresponding message input specified on that instance. For each message input from an instance, there must be a corresponding message output specified on that instance.
  6. A message output may not be causally dependent on its corresponding message input, directly or via other messages. This property is verified by constructing a partial order on communication events and checking that the directed graph obtained from this partial order does not contain cycles. A message event precedes all message events that follow it in an instance specification, and every message input is preceded by its corresponding message output.
  7. Only declared instances may be referenced in the shared instance list of a condition.
  8. A shared condition must appear equally many times in the instances sharing it.
  9. Only declared instances may be referenced in a process creation.
  10. There must not be more than one process creation event with a given instance name.
- A timeout or reset event can only occur after a corresponding timer set event, and a stop event must be the last on the time line. ∎

Now, we specify conditions for a HMSC to be well-formed.

context: WellformedBMSC

```
scheme WellformedHMSC =
  extend WellformedBMSC with
  class
    type HMSC = {| h : HMSC′ • wf_HMSC(h) |}

    value
      wf_HMSC : HMSC′ → Bool
      wf_HMSC(b, a, s, e) ≡
        /* 1 */
        dom a = dom b ∧
        /* 2 */
        (∀ bmscs : BMSC_Name-set • bmscs ∈ rng a ⇒ bmscs ⊆ dom a) ∧
        /* 3 */
        s ∈ dom b ∧
        /* 4 */
        e ⊆ dom b
  end
```

Annotations

- A HMSC is well-formed, if each of the following conditions hold:
- The set of arrows must emanate from BMSCs that are in the mapping of BMSC names to BMSCs.
- The set of arrows must terminate at BMSCs that are in the mapping of BMSC names to BMSCs.
- The start node must be in the mapping of BMSC names to BMSCs.
- The end nodes must be in the mapping of BMSC names to BMSCs. ▪

13.1.7 An Example: IEEE 802.11 Wireless Network

Example 13.10 *An IEEE 802.11 Wireless Network:* We bring in a large example, this time without shading. ▪

Description

We illustrate the use of MSCs by modelling the possible exchanges of frames between an access point and a station in an IEEE 802.11 wireless network [224].

We assume the wireless network is operating under the Distributed Coordination Function and that no frames are lost due to transmission errors or collisions. Also, we omit some frame subtypes used for power save functions, etc.

A station is any device that conforms to the physical layer and medium access control layer specifications in the IEEE 802.11 standard. An access

point is a station that additionally routes frames between the wireless network and some other network (usually a wired LAN). IEEE 802.11 uses the carrier sense multiple access/collision avoidance (CSMA/CA) technology for accessing the medium. The HMSC is shown in Fig. 13.11 and the referenced BMSCs in Fig. 13.12.

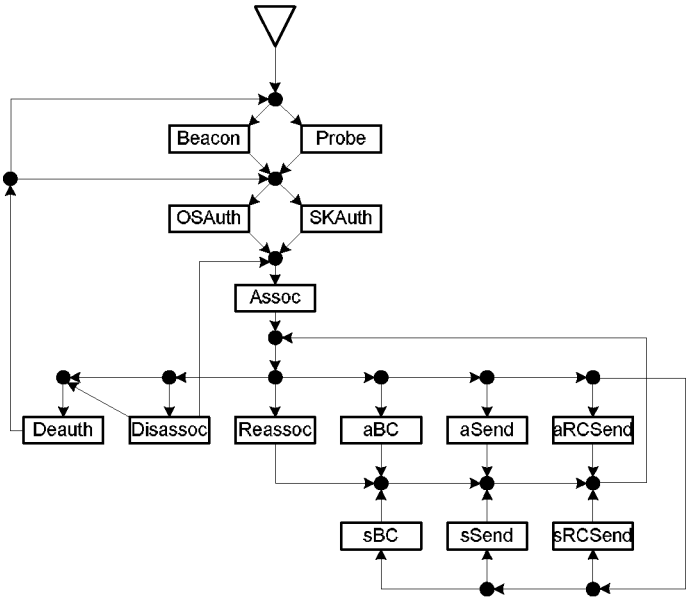

Fig. 13.11. HMSC of 802.11 wireless network with one access point and one station.

Initially, the station has no contact with the access point. It discovers the access point by scanning the available channels. Scanning may be either passive, in which case it waits for a beacon frame from the access point, or it may be active, in which case it emits probe frames. If an access point receives a probe frame it will respond with a probe response frame giving information (timing, SSID, etc.) necessary for joining the network. Once the station has contact with a network, it must be authenticated. In an 802.11 network there are two authentication methods: *open system* and *shared key*. In the former, any station requesting authentication may become authenticated. More specifically, the station will send an authentication request and the access point will respond with an authentication response. In the latter, only stations with knowledge of a shared secret key may become authenticated. In this case the authentication protocol consists of four messages.

First, the station sends an authentication request. The access point replies with a challenge message containing a nonce value. The station encrypts the nonce value using the shared secret key and returns it in an authentication response frame. Then the access point decrypts the received encrypted nonce

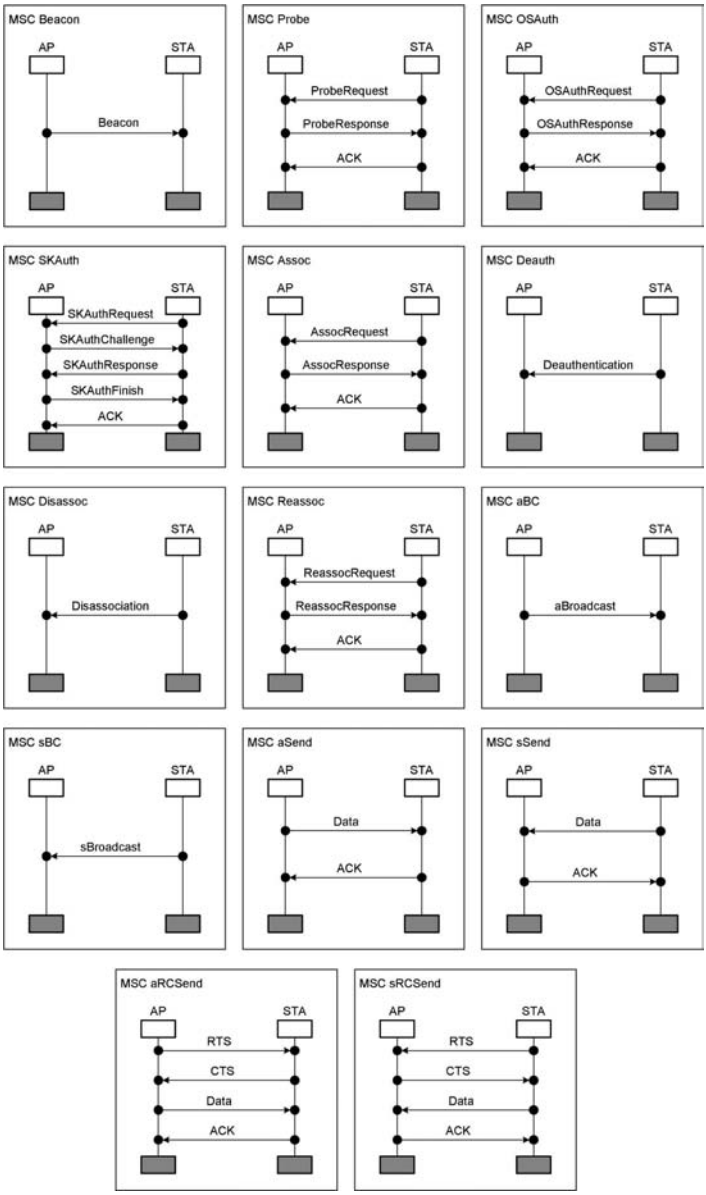

Fig. 13.12. BMSCs referenced in Fig. 13.11

and compares it with the original nonce. If they match the station is considered authenticated. The outcome of the comparison is sent to the station, confirming either that it is authenticated or that authentication failed.

The next step is for the station to become associated with the access point. Several 802.11 networks, each with their own acces point, may be joined to form an extended logical network, within which stations may move freely. Association is a means of recording which access point in such an extended network a given station is currently able to communicate with. Each station may be associated with only one access point at a time, while an access point may be associated with zero, one or more stations. An association is established by the station sending an association request frame to the access point it wishes to associate with. The access point replies with an association response frame.

An RSL Model of the IEEE 802.11 Example

Example 13.11 *An RSL Model of the IEEE 802.11 Example:* We now show an RSL model that conveys the same information as the MSC model, namely the sequence of messages that may be passed in the given 802.11 wireless network. We model the two entities as two concurrent processes which exchange messages by communicating on two channels. We do not take advantage of the features of RSL to describe the contents of the messages or how they are formed.

Text and formulas are not framed. ▪

First, we define the types of frames. In IEEE 802.11 there are three overall types of frames: data, management and control frames. Each type of frame has several subtypes.

```
scheme IEEE80211 =
  class
    type
      Frame = ManFrame | CtrFrame | DataFrame,
      ManFrame ==
        Beacon |
        ProbeRequest |
        ProbeResponse |
        OSAuthRequest |
        OSAuthResponse |
        SKAuthRequest |
        SKAuthChallenge |
        SKAuthResponse |
        SKAuthFinish |
        AssocRequest |
        AssocResponse |
        Deauthentication |
        Disassociation |
        ReassocRequest |
        ReassocResponse,
```

```
    CtrFrame == ACK | CTS | RTS,
    DataFrame == Data | Broadcast

  channel s_a : Frame, a_s : Frame
end
```

Annotations

- A *frame* is a *management, control,* or *data* frame.
- A *management* frame has one of 15 subtypes.
- A *control* frame has subtype *acknowledgement, clear-to-send,* or *request-to-send.*
- A *data* frame is a unicast *data* frame or a *broadcast* frame.
- There is a pair of channels between the access point and the stations. ∎

Now we describe the behaviour of the access point in terms of the communications in which it will participate. Note that received messages only serve to advance the communication, while the contents and type of message received is ignored. Also note that in situations where the access point may do one of several things we abstract this choice as a nondeterministic internal choice. The specification is not robust in the sense that the access point does not check that the messages received from the station are of the correct type and subtype.

context: IEEE80211

```
scheme IEEE80211_ap =
  extend IEEE80211 with
  class
    value
      AP : Unit → in s_a out a_s Unit
      AP() ≡ (a_beacon() ▯ a_probe()),

      a_beacon : Unit → in s_a out a_s Unit
      a_beacon() ≡ a_s!Beacon ; (a_osauth() ⫿ a_skauth()),

      a_probe : Unit → in s_a out a_s Unit
      a_probe() ≡
        let proberequest = s_a? in skip end ;
        a_s!ProbeResponse ;
        let ack = s_a? in skip end ;
        (a_osauth() ⫿ a_skauth()),

      a_osauth : Unit → in s_a out a_s Unit
      a_osauth() ≡
        let osauthrequest = s_a? in skip end ;
```

```
    a_s!OSAuthResponse ;
    let ack = s_a? in skip end ;
    a_assoc(),

a_skauth : Unit → in s_a out a_s Unit
a_skauth() ≡
    let skauthrequest = s_a? in skip end ;
    a_s!SKAuthChallenge ;
    let skauthresponse = s_a? in skip end ;
    a_s!SKAuthFinish ;
    let ack = s_a? in skip end ;
    a_assoc(),

a_assoc : Unit → in s_a out a_s Unit
a_assoc() ≡
    let assocrequest = s_a? in skip end ;
    a_s!AssocResponse ;
    let ack = s_a? in skip end ;
    a_op(),

a_op : Unit → in s_a out a_s Unit
a_op() ≡
    a_deauth()
    □
    a_disassoc()
    □
    a_reassoc()
    □
    a_abc()
    □
    a_asend()
    □
    a_arcsend()
    □
    a_sbc()
    □
    a_ssend()
    □
    a_srcsend(),

a_deauth : Unit → in s_a out a_s Unit
a_deauth() ≡
    let deauthentication = s_a? in skip end ;
    (a_osauth() ⫿ (a_skauth() □ AP())),
```

```
    a_disassoc : Unit → in s_a out a_s Unit
    a_disassoc() ≡
      let disassociation = s_a? in skip end ;
      (a_deauth() ▯ a_assoc()),

    a_reassoc : Unit → in s_a out a_s Unit
    a_reassoc() ≡
      let reassocrequest = s_a? in skip end ;
      a_s!ReassocResponse ;
      let ack = s_a? in skip end ;
      a_op(),

    a_sbc : Unit → in s_a out a_s Unit
    a_sbc() ≡ let broadcast = s_a? in skip end ; a_op(),

    a_ssend : Unit → in s_a out a_s Unit
    a_ssend() ≡ let data = s_a? in skip end ; a_s!ACK ; a_op(),

    a_srcsend : Unit → in s_a out a_s Unit
    a_srcsend() ≡
      let rts = s_a? in skip end ;
      a_s!CTS ;
      let data = s_a? in skip end ;
      a_s!ACK ;
      a_op(),

    a_abc : Unit → in s_a out a_s Unit
    a_abc() ≡ a_s!Broadcast ; a_op(),

    a_asend : Unit → in s_a out a_s Unit
    a_asend() ≡ a_s!Data ; let ack = s_a? in skip end ; a_op(),

    a_arcsend : Unit → in s_a out a_s Unit
    a_arcsend() ≡
      a_s!RTS ;
      let cts = s_a? in skip end ;
      a_s!Data ;
      let ack = s_a? in skip end ;
      a_op()
end
```

We now give the corresponding behaviour of the station. This is essentially the inverse of that of the access point. Again, choices are abstracted as internal nondeterminism.

context: IEEE80211_ap

```
scheme IEEE80211_sta =
  extend IEEE80211_ap with
  class
    value
      STA : Unit → in a_s out s_a Unit
      STA() ≡ (s_beacon() ▯ s_probe()),

      s_beacon : Unit → in a_s out s_a Unit
      s_beacon() ≡
        let beacon = a_s? in skip end ;
        (s_osauth() ▯ s_akauth()),

      s_probe : Unit → in a_s out s_a Unit
      s_probe() ≡
        s_a!ProbeRequest ;
        let proberesponse = a_s? in skip end ;
        s_a!ACK ;
        (s_osauth() ▯ s_akauth()),

      s_osauth : Unit → in a_s out s_a Unit
      s_osauth() ≡
        s_a!OSAuthRequest ;
        let osauthresponse = a_s? in skip end ;
        s_a!ACK ;
        s_assoc(),

      s_akauth : Unit → in a_s out s_a Unit
      s_akauth() ≡
        s_a!SKAuthRequest ;
        let skauthchallenge = a_s? in skip end ;
        s_a!SKAuthResponse ;
        let skauthfinish = a_s? in skip end ;
        s_a!ACK ;
        s_assoc(),

      s_assoc : Unit → in a_s out s_a Unit
      s_assoc() ≡
        s_a!AssocRequest ;
        let assocresponse = a_s? in skip end ;
        s_a!ACK ;
        s_op(),

      s_op : Unit → in a_s out s_a Unit
      s_op() ≡
```

```
    s_deauth()
    ||
    s_disassoc()
    ||
    s_reassoc()
    ||
    s_abc()
    ||
    s_asend()
    ||
    s_arcsend()
    ||
    s_abc()
    ||
    s_asend()
    ||
    s_arcsend(),

s_deauth : Unit → in a_s out s_a Unit
s_deauth() ≡
    s_a!Deauthentication ; ((s_osauth() [] s_akauth()) || STA()),

s_disassoc : Unit → in a_s out s_a Unit
s_disassoc() ≡ s_a!Disassociation ; (s_deauth() [] s_assoc()),

s_reassoc : Unit → in a_s out s_a Unit
s_reassoc() ≡
    s_a!ReassocRequest ;
    let reassocresponse = a_s? in skip end ;
    s_a!ACK ;
    s_op(),

s_sbc : Unit → in a_s out s_a Unit
s_sbc() ≡ s_a!Broadcast ; s_op(),

s_ssend : Unit → in a_s out s_a Unit
s_ssend() ≡ s_a!Data ; let ack = a_s? in skip end ; s_op(),

s_srcsend : Unit → in a_s out s_a Unit
s_srcsend() ≡
    s_a!RTS ;
    let cts = a_s? in skip end ;
    s_a!Data ;
    let ack = a_s? in skip end ;
    s_op(),
```

```
    s_abc : Unit → in a_s out s_a Unit
    s_abc() ≡ let broadcast = a_s? in skip end ; s_op(),

    s_asend : Unit → in a_s out s_a Unit
    s_asend() ≡ let data = a_s? in skip end ; s_a!ACK ; s_op(),

    s_arcsend : Unit → in a_s out s_a Unit
    s_arcsend() ≡
      let rts = a_s? in skip end ;
      s_a!CTS ;
      let data = a_s? in skip end ;
      s_a!ACK ;
      s_op()
end
```

This example has hopefully demonstrated the power of MSCs as a specification method. Clearly, the MSC specification is much more compact than the corresponding RSL specification, and it is also much more readable. The power of RSL, however, becomes apparent if one wants to add an additional layer of detail, for example, by adding parameters to the messages and explaining how parameters from incoming messages are related to the parameters of outgoing messages. While MSCs are good at specifying one aspect (namely sequences of events) of a system, RSL is expressive enough to specify many aspects.

13.1.8 Semantics of Basic Message Sequence Charts

We now give a semantics of BMSCs by defining an RSL function, *S*, that yields the possible traces of a given BMSC. A trace is a causally ordered sequence of events. Note that the semantics is in general nondeterministic, in the sense that a given BMSC may have many legal sequences of events.

```
scheme BMSC_Semantics =
  extend WellformedBMSC with
  class
    value
      S : BMSC → (Event*)-set
      S(bmsc) ≡
        {el|el:Event*•el ∈ interleave(bmsc)∧isValid(el,{})}

      interleave : BMSC → (Event*)-set
      interleave(bmsc) ≡
        interleave(⟨evtl(inst)|inst in instances(bmsc)⟩,⟨⟩)

      interleave : (Event*)* × (Event*)* → (Event*)-set
```

```
        interleave(evtll, evtll′) ≡
            if evtll = ⟨⟩
a             then {}
              else
                  let head = hd evtll in
                  (let rest = interleave(⟨tl head⟩⌢tl evtll⌢evtll′,⟨⟩) in
b                 {⟨hd head⟩⌢r|r:Event*•r ∈ rest} end)
c                 ∪ interleave(tl evtll,⟨head⟩⌢evtll′)
                  end
            end

        isValid : Event* × Msg_Name-set → Bool
        isValid(evtl, mnms) ≡
            case hd evtl of
                mk_Output(mnm,pars,adr) →
                    isValid(tl evtl,mnms ∪{mnm}),
                mk_Input(mnm,pars,adr) →
                    id ∈ ids ∧ isValid(tl evtl,mnms\{mnm})
            end
    end
```

Annotations

- The semantics of a BMSC, S(bmsc), is a set of lists of events, where each list is an interleaving of the events of each of the instances in the BMSC, and the set contains only those lists that are valid.
- The interleaving of a BMSC is an interleaving of the event lists of its instances.
  (a) The interleaving of an empty list of events is the empty set.
  (b) The interleaving of a non-empty list of event lists is obtained by selecting the head element of the head of the list and adding that element as the first element of all interleavings of the remaining event-lists,
  (c) and forming the union with the set of interleavings obtained from the rest of the list.
- An event list is valid if every input event causally follows its corresponding output event in the list. ▪

13.1.9 Semantics of High-Level Message Sequence Charts

We leave it as Exercise 13.3 for the reader to combine the above into functions which give a semantics of HMSCs.

13.2 Live Sequence Charts: Informal Presentation

13.2.1 Live Sequence Chart Syntax

In this section we informally describe the components of live sequence charts (LSC). We return to the question of a formal semantics of a subset of LSCs in Sect. 13.4.

Graphical Syntax of Live Sequence Charts

LSCs were proposed by Damm and Harel [89] as an extension of MSCs. They identified a number of shortcomings and weaknesses of the MSC standard and proposed a range of new concepts and notation to overcome these problems.

One of the major problems with the semantics of MSCs is that it is not clear whether an MSC describes all behaviours of a system or just a set of possible behaviours. Typically, the latter view would be used in early stages of development, while the former would apply in later stages when the behaviour is more fixed. Another problem noted by Damm and Harel is the inability of MSCs to specify liveness, i.e., MSCs have no constructions for enforcing progress. Damm and Harel also view the lack of semantics for conditions to be a problem.

Universal and Existential Charts

The most prominent feature of LSCs is the introduction of a distinction between optional and mandatory behaviour. This applies to several elements in charts. A distinction is introduced between *universal* charts and *existential* charts.

Universal charts specify behaviour that must be satisfied by every possible run of a system. This may be compared to universal quantification over the runs of the system. On the other hand, existential charts specify behaviour that must be satisfied by at least one run of the system. This is like existential quantification over the runs of the system. The typical application of existential charts would be in the early stages of the development process, particularly in domain modelling. An existential chart specifies a scenario that may be used to describe characteristic behaviours of the domain.

Universal charts would typically be used later in the development process, particularly in requirements engineering and in requirements documents. Universal charts are designated by a fully drawn box around the chart, while existential charts are designated by a dashed box.

Example 13.12 Figure 13.13 shows a universal LSC with two instances, A and B. The behaviour specified by this chart must (i.e., shall) be satisfied by every run of the system.

Figure 13.13 shows an existential LSC. This represents a scenario that at least one run of the system must satisfy.

The four messages of Fig. 13.13 are discussed in Example 13.14 below. ▪

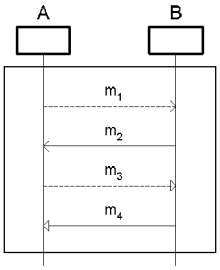

Fig. 13.13. Universal LSC

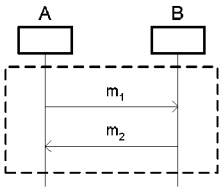

Fig. 13.14. Existential LSC

Precharts

LSC introduces the notion of a *prechart* to restrict the applicability of a chart. The prechart is like a precondition that when satisfied activates the main chart. A given system need only satisfy a universal chart whenever it satisfies the prechart. An empty prechart is satisfied by any system. A prechart can be considered as the expression in an IF statement where the body of the THEN part is the universal chart. The prechart is denoted by a dashed hexagon containing zero, one or more events.

Example 13.13 Figure 13.15 shows a universal LSC with a prechart consisting of the single message *activate*. In this case, the behaviour specified in the body of the chart only applies to those runs of the system where the message *activate* is sent from instance A to instance B. ▪

"Hot" and "Cold" Messages

LSC allow messages to be "hot" or "cold". A hot message is mandatory, i.e., if it is sent then it must be received eventually. This is denoted by a fully drawn arrow. For a cold message reception is not required, i.e., it may be "lost". This is denoted by a dashed arrow.

Synchronous and Asynchronous Messages

Also, a message may be specified as either synchronous or asynchronous. Synchronous messages are denoted by an open arrowhead $\Rightarrow$, while asynchronous messages are denoted by a closed arrowhead $\rightarrow$.

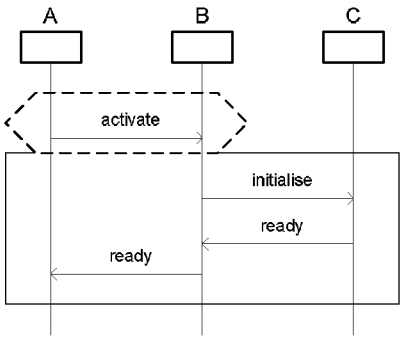

Fig. 13.15. Prechart

Example 13.14 Figure 13.13 illustrates the four kinds of messages: hot and cold, synchronous and asynchronous. Message m_1 is cold and synchronous. Message m_2 is hot and synchronous. Message m_3 is cold and asynchronous. Finally, message m_4 is hot and asynchronous. ▪

Conditions

In LSC conditions are promoted to first-class events. The difference is that conditions now have an influence on the execution of a chart, while in MSC they were merely comments. Again, a distinction is made between a hot (mandatory) condition, which, if evaluated to false, causes nonsuccessful termination of the chart, and a cold condition (optional) which, if evaluated to false, causes successful termination of the chart. A hot condition is like an invariant which must be satisfied.

By combining a prechart with a universal chart containing just a single hot condition that always evaluates to false, it is possible to specify forbidden scenarios, since the scenario expressed in the prechart will then always cause nonsuccessful termination. A shared condition forces synchronisation among the sharing instances, i.e., the condition will not be evaluated before all instances have reached it and no instance will progress beyond the condition until it has been evaluated.

Example 13.15 Figure 13.16 illustrates two conditions. The first is hot, while the second is cold. If the hot condition evaluates to false, the chart is aborted, indicating an erroneous situation. If the second condition evaluates to false, the current (sub)chart is exited successfully. ▪

Subcharts

Iteration and conditional execution are obtained by means of *subcharts*. Subcharts are LSCs that are specified for a subset of the instances of the containing

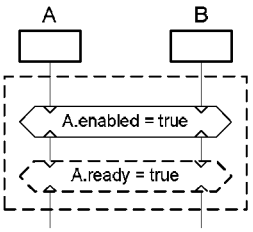

Fig. 13.16. Conditions

LSC and possibly additional new instances. Iteration is denoted by annotating the top-left corner of the chart with an integer constant for limited iteration or an asterisk for unlimited iteration. A subchart is exited successfully either when a limited iteration has executed the specified number of times, or when a cold condition evaluates to false.

By combining subcharts with cold conditions, WHILE and DO–WHILE loops may be created. Additionally, a special form of subchart with two parts is used to create an IF–THEN–ELSE construct. The first part of the subchart has a cold condition as the first event. If the condition evaluates to true, the first part of the subchart is executed. If the condition evaluates to false, the second part of the subchart is executed.

Example 13.16 Figure 13.17 illustrates limited iteration. Instance A will send the message m_1 60 times to instance B. ▪

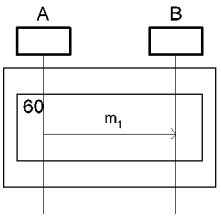

Fig. 13.17. Limited iteration

Example 13.17 Figure 13.18 illustrates unlimited iteration with a stop condition, essentially like a DO–WHILE loop. The message m_1 will be sent repeatedly until the condition becomes false. Once that happens, the subchart is exited. ▪

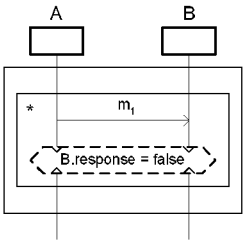

Fig. 13.18. DO–WHILE loop

Example 13.18 Figure 13.19 is similar to the previous situation, except that the condition is now checked before the first message is sent, thus mimicking a WHILE loop. ▪

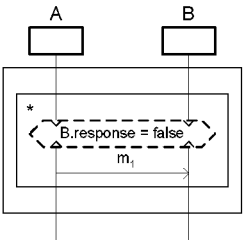

Fig. 13.19. WHILE loop

Example 13.19 Figure 13.20 is like Fig. 13.19 except that there is no iteration. Thus, the message m_1 will be sent once if the condition evaluates to true, and it will not be sent if the condition evaluates to false. Therefore, this construction is like an IF–THEN construct. ▪

Example 13.20 In Fig. 13.21 the special construction for IF–THEN–ELSE is illustrated. The two subcharts represent the THEN and ELSE branches. If the condition evaluates to true, the first subchart is executed, otherwise the second subchart is executed. In either case, the subchart not chosen is skipped entirely. ▪

Locations

The distinction between hot and cold is also applied to the timeline of an instance. Any point where an event is specified on the timeline is called a

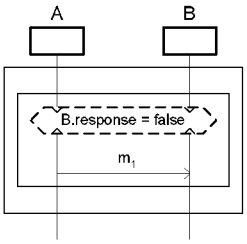

Fig. 13.20. IF–THEN conditional

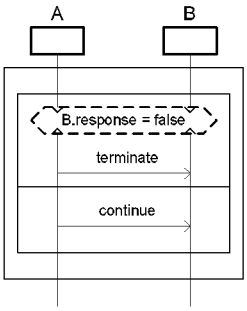

Fig. 13.21. IF–THEN–ELSE conditional

location. A location may be hot indicating that the corresponding event must eventually take place, or cold indicating that event might never occur. A hot location is represented by the time line being fully drawn, while a cold location is represented by a dashed time line. The timeline may alternate between being fully drawn and dashed.

The addition of cold locations conflicts with the representation of coregions inherited from MSCs. For this reason, the syntax for a coregion is modified to be a dashed line positioned next to the part of the time line that the coregion spans.

Example 13.21 Figure 13.22 illustrates the syntax for optional progress. The timeline is fully drawn at the location where the message m_1 is sent and received, indicating that these events must eventually take place. This guarantees liveness. At the location where the message m_2 is sent and received, the time line is dashed, indicating that neither instance is required to progress to the sending or receiving of m_2. If an instance does not progress beyond a location l, then no event on the time line of that instance following l will take place. Thus, in this case, if m_2 is never sent, m_3 will never be sent. ▪

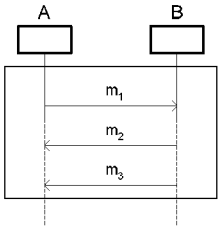

Fig. 13.22. Optional progress

13.2.2 A Live Sequence Chart Example, I

Example 13.22 *A Live Sequence Chart, Part I:* We conclude this section with an example. This example is concluded by Example 13.23 in Sect. 13.4.3. We omit shading. ▪

Figure 13.23 shows an example LSC with three instances. The first step is to convert the graphical syntax into the textual syntax. The result is shown below.

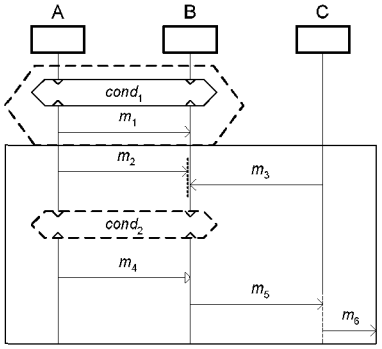

Fig. 13.23. Example live sequence chart

lsc Example;
  instance *A*
    prechart
      hot hotcondition($cond_1$) ;
      hot out m_1 **to** *B* **async** ;
    end prechart body hot out m_2 **to** *B* **async** ;
      hot coldcondition($cond_2$) ;
      hot out m_4 **to** *B* **sync** ;
    end body

```
    end instance
    instance B
        prechart
            hot hotcondition(cond1) ;
            hot in m1 from A async ;
        end prechart
        body
            hot concurrent
                in m2 from A async ;
                in m3 from C async ;
            end concurrent ;
            hot coldcondition(cond2) ;
            hot in m4 from A sync ;
            hot out m5 to C async ;
        end body
    end instance
    instance C
        body
            hot out m3 to B async ;
            cold in m5 from B async ;
            cold out m6 to env async ;
        end body
    end instance
end lsc
```

13.3 Process Algebra

The ITU standard Z.120 for MSCs includes a formal algebraic semantics based on the process algebra PA_ϵ introduced by Baeten and Weijland [27]. In this section we first briefly review the definition of PA_ϵ following [326] and [26], and then present an extension of that algebra (named PAc_ϵ), which will be used for defining the semantics of a subset of LSCs in Section 13.4.2 and for expressing communication behaviours of RSL specifications in Sect. 13.5.2.

The material in this section cannot be considered to belong to the field of software engineering. Rather, it belongs to the field of computer science. The reader whose interest is mainly focused on the application of MSCs and LSCs to actual engineering problems may skip the rest of this chapter. Those who wish to gain a deeper understanding of the relations between sequence charts and RSL are encouraged to read on.

The material that follows only scratches the surface of the topic of process algebras. The theoretical foundations for the process algebras presented here are given in [317].

13.3.1 The Process Algebra PA_ϵ

The algebraic theory of PA_ϵ is given as an equational specification $(\Sigma_{PA_\epsilon}, E_{PA_\epsilon})$, consisting of the signature, Σ_{PA_ϵ}, and a set of equations, E_{PA_ϵ}. We first define the signature and equations and then give the intuition behind the definitions.

Signature

The one-sorted signature, Σ_{PA_ϵ}, consists of

1. two special constants δ and ϵ
2. a set of unspecified constants A, for which $\{\delta, \epsilon\} \cap A = \emptyset$
3. the unary operator $\surd$
4. the binary operators $+$, $\cdot$, $\|$ and $\between$

The unspecified set A is a parameter of the theory. Thus, applications of the theory require the theory to be instantiated with a specific set A. When the theory is applied to MSCs, the set A consists of identifiers for the atomic events of the chart.

For convenience and following tradition, we will apply the binary operators in infix notation, i.e., instead of $+(x, y)$ we will write $x + y$. To reduce the need for parentheses, operator precedences are introduced. The $\cdot$ operator binds strongest, the $+$ operator binds weakest.

Let V be a set of variables. Then terms over the signature Σ_{PA_ϵ} with variables from V, denoted $T(\Sigma_{PA_\epsilon}, V)$, are given by the inductive definition

1. $v \in V$ is a term.
2. $a \in A$ is a term.
3. δ is a term.
4. ϵ is a term.
5. If t is a term, then $\surd(t)$ is a term.
6. If t_1 and t_2 are terms, then $t_1 \, op \, t_2$ is a term, for $op \in \{+, \cdot, \|, \between\}$.

A term is called *closed* if it contains no variables. The set of closed terms over Σ_{PA_ϵ} is denoted $T(\Sigma_{PA_\epsilon})$.

Equations

The equations of PA_ϵ are of the form $t_1 = t_2$, where $t_1, t_2 \in T(\Sigma_{PA_\epsilon}, V)$. For $a \in A$ and $x, y, z \in V$ the equations, E_{PA_ϵ}, are given in Table 13.1.

The special constant δ is called *deadlock*. It denotes the process that has stopped executing actions and can never resume. The special constant ϵ is called the *empty process*. It denotes the process that terminates successfully without executing any actions. The elements of the set A are called *atomic actions*. These represent processes that cannot be decomposed into smaller parts. As mentioned above, the set A is given a concrete definition when

Table 13.1. Equations of PA_ϵ

$$x + y = y + x \quad \text{(A1)}$$
$$(x + y) + z = x + (y + z) \quad \text{(A2)}$$
$$x + x = x \quad \text{(A3)}$$
$$(x + y) \cdot z = x \cdot z + y \cdot z \quad \text{(A4)}$$
$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \quad \text{(A5)}$$
$$x + \delta = x \quad \text{(A6)}$$
$$\delta \cdot x = \delta \quad \text{(A7)}$$
$$x \cdot \epsilon = x \quad \text{(A8)}$$
$$\epsilon \cdot x = x \quad \text{(A9)}$$

$$x \parallel y = x \between y + y \between x + \surd(x) \cdot \surd(y) \quad \text{(F1)}$$
$$\epsilon \between x = \delta \quad \text{(F2)}$$
$$\delta \between x = \delta \quad \text{(F3)}$$
$$a \cdot x \between y = a \cdot (x \parallel y) \quad \text{(F4)}$$
$$(x + y) \between z = x \between z + y \between z \quad \text{(F5)}$$

$$\surd(\epsilon) = \epsilon \quad \text{(T1)}$$
$$\surd(\delta) = \delta \quad \text{(T2)}$$
$$\surd(a \cdot x) = \delta \quad \text{(T3)}$$
$$\surd(x + y) = \surd(x) + \surd(y) \quad \text{(T4)}$$

the theory is applied. For example, in defining the semantics of MSCs, the set A will contain the symbols that identify the events in the chart, such as $in(a, b, m1)$ identifying the event of instance b receiving message $m1$ from instance a.

The binary operators $+$ and $\cdot$ are called *alternative* and *sequential composition*, respectively. The alternative composition of processes x and y is the process that behaves as either x or y, but not both. The sequential composition of processes x and y is the process that first behaves as x until it reaches a terminated state and then behaves as y.

The binary operator $\parallel$ is called the *free merge*. The free merge of processes x and y is the process that executes an interleaving of the actions of x and y. The unary *termination operator* $\surd$ indicates whether the process it is applied to may terminate immediately. The termination operator is an auxiliary operator needed to define the free merge. The binary operator $\between$ is called the *left*

merge and denotes the process that executes the first atomic action of the left operand followed by the interleaving of the remainder of the left operand with the right operand. Like the termination operator, the left merge operator is an auxiliary operator needed to define free merge.

To see why the termination operator is necessary, consider Eq. F1. What happens in the free merge is that all possible sequences of atomic actions from the two operands are generated. When both operands become the empty process, we want the free merge to be the empty process as well, i.e., we want the equation $\epsilon \parallel \epsilon = \epsilon$ to hold. Because of Eq. F2, the two first alternatives in F1 become deadlock. However, the last alternative becomes the empty process, because of Eq. T1. Thus, with Eq. A6 we get the desired result. It is possible to give a simpler definition of the free merge without using the empty process or the termination operator, see [26], but for our purposes we need the empty process.

Derivability

We now define what it means for a term to be derivable from an equational specification. First, the two auxiliary notions of a substitution and a context are introduced.

Definition 13.1. A substitution $\sigma : V \to T(\Sigma, V)$ replaces variables with terms over Σ. The extension of σ to terms over Σ, denoted $\bar{\sigma} : T(\Sigma, V) \to T(\Sigma, V)$, is given by

1. $\bar{\sigma}(\delta) = \delta$
2. $\bar{\sigma}(\epsilon) = \epsilon$
3. $\bar{\sigma}(a) = a \quad \text{for } a \in A$
4. $\bar{\sigma}(v) = \sigma(v) \quad \text{for } v \in V$
5. $\bar{\sigma}(\surd(x)) = \surd(\bar{\sigma}(x))$
6. $\bar{\sigma}(x \ op \ y) = \bar{\sigma}(x) \ op \ \bar{\sigma}(y) \quad \text{for } op \in \{+, \cdot, \parallel, \between\}$

A substitution that replaces all variables with variable-free terms, i.e., closed terms, is called *closed.* ∎

Definition 13.2. A Σ context is a term $C \in T(\Sigma, V \cup \{\Box\})$, containing exactly one occurrence of the distinguished variable $\Box$. The context is written $C[\ ]$ to suggest that C should be considered as a term with a hole in it. Substitution of a term $t \in T(\Sigma, V)$ in $C[\ ]$ gives the term $C[\Box \mapsto t]$, written $C[t]$. ∎

Definition 13.3. *Let (Σ, E) be an equational specification and let t, s and u be arbitrary terms over Σ. The derivability relation, $\vdash$, is then given by the following inductive definition.*

$$
\begin{aligned}
& s = t \in E \quad \Rightarrow \quad (\Sigma, E) \vdash s = t \\
& (\Sigma, E) \vdash t = t \\
& (\Sigma, E) \vdash s = t \quad \Rightarrow \quad (\Sigma, E) \vdash t = s \\
& (\Sigma, E) \vdash s = t \quad \wedge \quad (\Sigma, E) \vdash t = u \quad \Rightarrow \quad (\Sigma, E) \vdash s = u \\
& (\Sigma, E) \vdash s = t \quad \Rightarrow \quad (\Sigma, E) \vdash \bar{\sigma}(s) = \bar{\sigma}(t) \quad \textit{for any substitution } \sigma \\
& (\Sigma, E) \vdash s = t \quad \Rightarrow \quad (\Sigma, E) \vdash C[s] = C[t] \quad \textit{for any context } C[-]
\end{aligned}
$$

If $(\Sigma, E) \vdash s = t$, *abbreviated* $E \vdash s = t$, *then the equation* $s = t$ *is said to be* derivable *from the equational specification* (Σ, E). ∎

Reduction to Basic Terms

We now venture deeper into the theory of process algebra and term rewriting systems. The goal is to show that there exists a model of the equational specification for PA_ϵ and that the equations E_{PA_ϵ} form a complete axiomatisation, i.e., that whenever two terms are equal in the model, then they are provably equal using the equations.

The first step is to show that any PA_ϵ term can be reduced to an equivalent so-called basic term consisting of only atomic actions, δ, ϵ, $+$ and $\cdot$. This result makes subsequent proofs easier, because we need only consider these simpler terms.

Definition 13.4. δ and ϵ are basic terms. An atomic action $a \in A$ is a *basic term.* If $a \in A$ and t is a basic term, then $a \cdot t$ is a basic term. If t and s are basic terms, then $t + s$ is a basic term. ∎

The next step is to show that any PA_ϵ term can be reduced to a basic term. To do this, a term rewriting system is defined.

Definition 13.5. A *term rewriting system* is a pair (Σ, R) of a signature, Σ, and a set, R, of rewriting rules. A rewriting rule is of the form $s \to t$, where $s, t \in T(\Sigma, V)$ are open terms over Σ, such that s is not a variable and $vars(t) \subseteq vars(s)$, where $vars(t)$ denotes the set of variables in the term t.

The one-step reduction relation, $\to$, is the smallest relation containing the rules, R, that is closed under substitutions and contexts. ∎

Definition 13.6. A term s is in *normal form* if there does not exist a term t, such that $s \to t$. A term s is called *strongly normalising* if there exist no infinite sequences of rewritings starting with s:

$$s \to s_1 \to s_2 \to \ldots$$

A term reduction system is called strongly normalising if every term in the system is strongly normalising. ∎

Table 13.2. Term rewriting system for PA_ϵ

$$x + x \to x \tag{RA3}$$
$$(x + y) \cdot z \to x \cdot z + y \cdot z \tag{RA4}$$
$$(x \cdot y) \cdot z \to x \cdot (y \cdot z) \tag{RA5}$$
$$x + \delta \to x \tag{RA6}$$
$$\delta \cdot x \to \delta \tag{RA7}$$
$$x \cdot \epsilon \to x \tag{RA8}$$
$$\epsilon \cdot x \to x \tag{RA9}$$

$$x \parallel y \to x \between y + y \between x + \surd(x) \cdot \surd(y) \tag{RF1}$$
$$\epsilon \between x \to \delta \tag{RF2}$$
$$\delta \between x \to \delta \tag{RF3}$$
$$a \cdot x \between y \to a \cdot (x \parallel y) \tag{RF4}$$
$$a \between x \to a \cdot x \tag{RF4'}$$
$$(x + y) \between z \to x \between z + y \between z \tag{RF5}$$

$$\surd(\epsilon) \to \epsilon \tag{RT1}$$
$$\surd(\delta) \to \delta \tag{RT2}$$
$$\surd(a \cdot x) \to \delta \tag{RT3}$$
$$\surd(x + y) \to \surd(x) + \surd(y) \tag{RT4}$$

The term rewriting system for PA_ϵ is shown in Table 13.2. Essentially, a term rewriting system is a collection of equations, that can be applied only one way. Compared with the equations of PA_ϵ in Table 13.1, there are no rewrite rules corresponding to A1 and A2, because these equations have no clear direction. Also, having a rule for A1 would render the rewrite system non-terminating.

A common method for proving normalisation of a term rewriting system is to define a partial ordering on the operators and constants of the signature Σ, and then extend this ordering to terms over Σ. There are several ways to define this extension. For our purposes, the so-called lexicographical variant of the recursive path ordering will suffice. The main reference for the following material is [26]. Other references are [27, 95, 251, 267].

Definition 13.7. Let $s, t \in T(\Sigma, V)$. We write $s >_{lpo} t$ if $s \to^+ t$, where $\to^+$ is the transitive closure of the reduction relation $\to$ defined by the rules RPO1-5 and LPO in Table 13.3. ∎

Table 13.3. Reduction rules

Reduction rules for lexicographical variant of recursive partial ordering

- RPO1. Mark head symbol ($k \geq 0$):
  $H(t_1, \ldots, t_k) \rightarrow H^*(t_1, \ldots, t_k)$
- RPO2. Make copies under smaller head symbol ($H > G$, $k \geq 0$):
  $H^*(t_1, \ldots, t_k) \rightarrow G(H^*(t_1, \ldots, t_k), \ldots, H^*(t_1, \ldots, t_k))$
- RPO3. Select argument ($k \geq 1$, $1 \leq i \leq k$):
  $H^*(t_1, \ldots, t_k) \rightarrow t_i$
- RPO4. Push $*$ down ($k \geq 1$, $l \geq 0$):
  $H^*(t_1, \ldots, G(s_1, \ldots, s_l), \ldots, t_k) \rightarrow H(t_1, \ldots, G^*(s_1, \ldots, s_l), \ldots, t_k)$
- RPO5. Handling contexts:
  $s \rightarrow t \quad \Rightarrow \quad H(\ldots, s, \ldots) \rightarrow H(\ldots, t, \ldots)$
- LPO. Reduce ith argument ($k \geq 1$, $1 \leq i \leq k$, $l \geq 0$,
  H has lexicographical status wrt. the ith argument):
  Let $t \equiv H^*(t_1, \ldots, t_{i-1}, G(s_1, \ldots, s_l), t_{i+1}, \ldots, t_k)$
  then $t \rightarrow H(t, \ldots, t, G^*(s_1, \ldots, s_l), t, \ldots, t)$

Theorem 13.8. *Strong Normalisation (I)* (Kamin and Lévy [259]). Let (Σ, R) be a term rewriting system with finitely many rewrite rules and let $>$ be a well-founded partial ordering on Σ. If $s >_{lpo} t$ for each rewriting rule $s \rightarrow t \in R$, then the term rewriting system (Σ, R) is strongly normalising. ∎

Proof. See [259]. ∎

The intuition behind Theorem 13.8 is that if $x >_{lpo} y$, then y is a less complicated term than x, where we consider basic terms to be the simplest and general terms to be the most complicated. Thus, if all the rules can only make terms less complicated, we are bound to eventually reach a term that can not be simplified.

Lemma 13.9. *Strong Normalisation (II)* The term rewriting system for PA_ϵ in Table 13.2 is strongly normalizing. ∎

Proof. According to Theorem 13.8, it is sufficient to define a partial ordering on Σ_{PA_ϵ} and show that each rewriting rule satisfies the extension of the ordering to $T(\Sigma)$. We use the partial order $\| > \between > \surd > \cdot > + > \epsilon > \delta$. $\cdot$ has lexicographical status with regard to the first argument. Below, we illustrate the derivation for rewrite rules RA4 and RA5. The remaining derivations are given in [316].

$$
\begin{array}{lll}
(x + y) \cdot z & >_{lpo} & (x + y) \cdot^* z \qquad \text{RPO1} \\
& >_{lpo} & (x + y) \cdot^* z + (x + y) \cdot^* z \qquad \text{RPO2} \\
& >_{lpo} & (x +^* y) \cdot z + (x +^* y) \cdot z \qquad \text{RPO4, RPO5} \\
& >_{lpo} & x \cdot z + y \cdot z \qquad \text{RPO3, RPO5}
\end{array}
$$

$$
\begin{array}{lll}
(x \cdot y) \cdot z & >_{lpo} & (x \cdot y) \cdot^* z & \text{RPO1} \\
& >_{lpo} & (x \cdot^* y) \cdot ((x \cdot y) \cdot^* z) & \text{LPO} \\
& >_{lpo} & x \cdot ((x \cdot^* y) \cdot z) & \text{RPO3, RPO5, RPO5} \\
& >_{lpo} & x \cdot (y \cdot z) & \text{RPO3, RPO5}
\end{array}
$$

Thus, the term rewriting system for PA_ϵ is strongly normalising. ∎

We are now ready to prove that every PA_ϵ term has an equivalent basic term.

Theorem 13.10. For every PA_ϵ term, s, there is a corresponding basic term, t, such that $PA_\epsilon \vdash s = t$. ∎

Proof. By the *strong normalisation (II)* theorem the term rewriting system for PA_ϵ is strongly normalizing. Thus, for every term t, there is a finite sequence of rewritings

$$t \to t_1 \to t_2 \to \cdots \to s$$

where s is in normal form.

We use a proof by contradiction to show that s cannot contain $\|$, $\between$ or $\surd$. Assume, therefore, that s is in normal form and that $s = C[x \parallel y]$. But then the rewriting RF1 can be used, thus contradicting that s is in normal form. Now assume that s is in normal form and that $s = C[x \between y]$. Then there are three cases

- $x = u \between w$: in this case we can use the argument recursively to show that u or one of its sub-terms can be reduced by a rewrite rule. This line of reasoning is valid since we deal with finite terms.
- $x = \surd(u)$: in this case either x can be rewritten using one of RT1-4, or we can apply the whole argument to u to show that some sub-term of u can be rewritten.
- in all other cases one of the four rewrite rules RF2-4 may be applied to s, thus forming a contradiction.

Finally, we can use the same argument as above to show that if $s = C[\surd(x)]$ then either we can use one of the rewriting rules RT1-4 on s directly, or some sub-term of x can be reduced using a rewrite rule.

Thus, in all cases we have a contradiction and the theorem follows. ∎

13.3.2 Semantics of PA_ϵ

We now proceed to define a semantics for PA_ϵ. See Table 13.4.

We use a structural operational semantics in the style of Plotkin [402]. Based on the semantics, we define a behavioural equivalence on PA_ϵ terms, called bisimulation equivalence. We then show that the quotient algebra of PA_ϵ terms under bisimulation equivalence is a model of the equational specification

PA_ϵ, which implies soundness of the equations. Finally, we prove completeness of the equations.

A Plotkin-style operational semantics is defined using a set of derivation rules. For our purpose, the premises and conclusion of a derivation rule are formulas of either the form

$$x \xrightarrow{a} x'$$

or of the form

$$x \downarrow$$

Informally, the former formula means that process x can evolve into process x' by performing action a. The latter formula means that process x can terminate immediately and successfully.

A formula ϕ is provable from a set of deduction rules, if there is a rule

$$\frac{\varphi_1 \quad \varphi_2 \quad \cdots \quad \varphi_n}{\varphi}$$

such that there exists a substitution $\sigma : V \to T(\Sigma, V)$ satisfying $\sigma(\varphi) = \phi$ and if $\sigma(\varphi_i)$ is provable from the deduction rules for $i = 1, 2, \ldots, n$.

The deduction rules of the operational semantics for PA_ϵ are shown in Table 13.4. An empty premise is designated by a $\Box$ above the line.

Table 13.4. Structural operational semantics of PA_ϵ

$\dfrac{\Box}{a \xrightarrow{a} \epsilon}$	Act
$\dfrac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	Cho1
$\dfrac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$	Cho2
$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	Seq1
$\dfrac{x \downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$	Seq2
$\dfrac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$	Par1
$\dfrac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$	Par2
$\dfrac{x \xrightarrow{a} x'}{x \between y \xrightarrow{a} x' \parallel y}$	Lme
$\dfrac{\Box}{\epsilon \downarrow}$	EpT
$\dfrac{x \downarrow}{x + y \downarrow}$	ChoT1
$\dfrac{y \downarrow}{x + y \downarrow}$	ChoT2
$\dfrac{x \downarrow \quad y \downarrow}{x \cdot y \downarrow}$	SeqT
$\dfrac{x \downarrow \quad y \downarrow}{x \parallel y \downarrow}$	ParT
$\dfrac{x \downarrow \quad y \downarrow \quad x \xrightarrow{a} x'}{x \between y \downarrow}$	LmeT
$\dfrac{x \downarrow}{\surd(x) \downarrow}$	TerT

We seek a means of identifying terms that behave "in the same way". This form of behavioural equivalence is captured in the notion of *bisimulation*. Here, we use the strong formulation of bisimulation, due to Park [387].

Definition 13.11. *Strong bisimulation equivalence* $\sim \subseteq T(\Sigma) \times T(\Sigma)$, is the largest symmetric relation, such that for all $x, y \in T(\Sigma)$, if $x \sim y$, then the following conditions hold

1. $\forall x' \in T(\Sigma) : x \xrightarrow{a} x' \Rightarrow \exists y' \in T(\Sigma) : y \xrightarrow{a} y' \wedge x' \sim y'$
2. $x \downarrow \Leftrightarrow y \downarrow$

Two terms, x and y, are called *bisimilar*, if there exists a bisimulation relation, $\sim$, such that $x \sim y$. ∎

It follows from the definition that the bisimulation relation is an equivalence relation, since it is reflexive, symmetric and transitive.

The next step is to show that the bisimulation relation is a congruence. Having established this result, it is easy to show that the deduction system in Table 13.4 is a model of the equational specification PA_ϵ. This is the same as saying that the equations for PA_ϵ are sound.

Definition 13.12. *(Congruence)* Let R be an equivalence relation on $T(\Sigma)$. R is called a congruence if for all n-ary function symbols $f \in \Sigma$

$$x_1 R y_1 \wedge \ldots \wedge x_n R y_n \Rightarrow f(x_1, \ldots, x_n) R f(y_1, \ldots, y_n)$$

where $x_1, \ldots, x_n, y_1, \ldots, y_n \in T(\Sigma)$. ∎

Definition 13.13. *(Baeten and Verhoef [25])* Let $T = (\Sigma, D)$ be a term deduction system and let $D = D(T_p, T_r)$, where T_p are the rules for the predicate (here $\downarrow$) and T_r are the rules for the relation (here $\xrightarrow{a}$). Let I and J be index sets of arbitrary cardinality, let $t_i, s_j, t \in T(\Sigma, V)$ for all $i \in I$ and $j \in J$, let $P_j, P \in T_p$ be predicate symbols for all $j \in J$, and let $R_i, R \in T_r$ be relation symbols for all $i \in I$. A deduction rule $d \in D$ is in *path formal* if it has one of the following four forms

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{f(x_1, \ldots, x_n) R t}$$

with $f \in \Sigma$ an n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{x R t}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{P f(x_1, \ldots, x_n)}$$

with $f \in \Sigma$ and n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables or

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{Px}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables.

A term deduction system is said to be in *path* format if all its deduction rules are in *path* format. ∎

Theorem 13.14. *(Baeten and Verhoef [25], Fokkink [117])* Let $T = (\Sigma, D)$ be a term deduction system. If T is in *path* format, then strong bisimulation equivalence is a congruence for all function symbols in Σ. ∎

Proof. See [25]. ∎

Lemma 13.15. Let T_{PA_ϵ} be the term deduction system defined in Table 13.4. Bisimulation equivalence is a congruence on the set of closed PA_ϵ terms. ∎

Proof. We show that the deduction rules EpT and Cho1 are in path format. Writing $\downarrow$ in non-fix notation, deduction rule EpT can be rewritten to

$$\frac{\{\ \}}{\downarrow(\epsilon)}$$

which is in the third form in Definition 13.13. Similarly, Cho1 can be rewritten to

$$\frac{\{x \xrightarrow{a} x'\}}{x + y \xrightarrow{a} x'}$$

which is in the first form.

It is easily verified that the remaining deduction rules are also in *path* format, so the lemma follows from Theorem 13.14. ∎

Having established that bisimulation equivalence is a congruence, we can construct the term quotient algebra $T(\Sigma_{PA_\epsilon})/\sim$. The reason we want to construct the quotient algebra is that it is an initial algebra, which is characterised by being the smallest algebra that captures the properties of the specification.

Recall that given an algebra A with signature Σ, the quotient algebra under the congruence $\equiv$, written $A/\equiv$ is defined as

- The carrier set of $A/\equiv$ consists of the equivalence classes of the carrier set of A under the equivalence relation $\equiv$, i.e., $|A/\equiv| = \{\ [x]_\equiv \mid x \in |A|\ \}$, where $[x]_\equiv = \{\ y \mid y \in |A| \wedge x \equiv y\ \}$.
- For each n-ary function symbol f_A in A, there is a corresponding n-ary function symbol $f_{A/\equiv}$ in $A/\equiv$, defined by

$$f_{A/\equiv}([x_1]_\equiv, \ldots, [x_n]_\equiv) = [f_A(x_1, \ldots, x_n)]_\equiv$$

Theorem 13.16. The set of closed PA_ϵ terms modulo bisimulation equivalence, notation $T(\Sigma_{PA_\epsilon})/\sim$, is a model of PA_ϵ. ∎

Proof. Recall that a Σ-algebra, A, is a model of an equational specification (Σ, E), if $A \models E$, i.e., if every equation derivable from E holds in A. Because bisimulation equivalence on PA_ϵ terms is a congruence by Lemma 13.15, it is sufficient to separately verify the soundness of each axiom in E_{PA_ϵ}, i.e., to show if $PA_\epsilon \vdash x = y$, then $x \sim y$.

We illustrate the procedure by verifying equation A1. We have to show that there exists a bisimulation equivalence $\sim_*$ such that $x + y \sim_* y + x$. Let $\sim_*$ be defined as $\{ (x+y, y+x) \mid x, y \in T(\Sigma_{PA_\epsilon}) \} \cup \{ (x, x) \mid x \in T(\Sigma_{PA_\epsilon}) \}$. Clearly, $\sim_*$ is symmetric. We now check the first bisimulation condition. $x+y$ can evolve only by following one of the two deduction rules Cho1 and Cho2. Suppose $x \xrightarrow{a} x'$, then $x + y \xrightarrow{a} x'$, but then we also have $y + x \xrightarrow{a} x'$. By definition $x' \sim_* x'$, so the condition is satisfied in this case. The symmetric case $y \xrightarrow{a} y'$ follows from the same argument. Next, the second bisimulation condition must be checked. Suppose $x \downarrow$, then by ChoT1 $x + y \downarrow$. But in that case by ChoT2 $y + x \downarrow$. Again the symmetric case $y \downarrow$ follows immediately.

The above procedure can be applied to the remaining equations to show that equal terms are bisimilar. Thus, the theorem follows. ∎

Finally, we show that PA_ϵ is a complete axiomatisation of the set of closed terms modulo bisimulation equivalence, i.e., whenever $x \sim y$, then $PA_\epsilon \vdash x = y$.

Theorem 13.17. The axiom system PA_ϵ is a complete axiomatisation of the set of closed terms modulo bisimulation equivalence. ∎

Proof. Due to Theorems 13.16 and 13.10 it suffices to prove the theorem for basic terms. The proof for basic terms is given in [26]. ∎

13.3.3 The Process Algebra PAc_ϵ

The process algebra PA_ϵ introduced in the previous section is sufficiently expressive to define the semantics of MSCs. However, the extension to LSCs calls for the introduction of an additional operator.

In this subsection we introduce the extended process algebra, called PAc_ϵ, for process algebra with conditional behaviour. PAc_ϵ is a conservative extension of PA_ϵ, meaning that the theory of PA_ϵ also holds in PAc_ϵ. We give an axiom system and a model of PAc_ϵ, and show that the axiom system is sound and complete. Our task now is considerably easier, since most of the results for PA_ϵ can be directly transferred to PAc_ϵ.

The signature of PAc_ϵ, Σ_{PAc_ϵ}, consists of

1. two special constants δ and ϵ

Table 13.5. Additional equations of PAc_ϵ

$\epsilon \triangleright x = x$	C1
$\delta \triangleright x = \epsilon$	C2
$x + y \triangleright z = (x \triangleright z) + (y \triangleright z)$	C3
$a \cdot x \triangleright y = a \cdot (x \triangleright y) + \bar{a}$, where $\bar{a} \in A \setminus \{a\}$	C4

2. a set of unspecified constants A, for which $\{\delta, \epsilon\} \cap A = \emptyset$
3. the unary operator $\surd$
4. the binary operators $+$, $\cdot$, $\|$, $\lfloor\!\lfloor$ and $\triangleright$

The binary operator $\triangleright$ is the *conditional behaviour* operator. The conditional behaviour of processes x and y is the process that either terminates successfully or executes x followed by y. The other operators and constants have the same meaning as they do in PA_ϵ.

Table 13.5 lists the additional equations E_{PAc_ϵ} for $a \in A$ and $x, y, z \in V$.

Table 13.6. Additional term rewriting rules for PAc_ϵ

$\epsilon \triangleright x \rightarrow x$	RC1
$\delta \triangleright x \rightarrow \epsilon$	RC2
$x + y \triangleright z \rightarrow x \triangleright z + y \triangleright z$	RC3
$a \cdot x \triangleright y \rightarrow a \cdot (x \triangleright y) + \bar{a}$	RC4
$a \triangleright y \rightarrow a \cdot y + \bar{a}$	RC4'

Theorem 13.18. The term rewriting system for PAc_ϵ in Table 13.6 is strongly normalizing. ∎

Proof. The proof is based on the proof of theorem 13.9. We add the conditional operator to the partial ordering: $\triangleright > \| > \lfloor\!\lfloor > \surd > \cdot > + > \epsilon > \delta$. We now show that the additional rewrite rules for PAc_ϵ satisfy the extension of the partial ordering to terms.

$\epsilon \triangleright x$	$>_{lpo}$	$\epsilon \triangleright^* x$	RPO1
	$>_{lpo}$	ϵ	RPO3

$$\begin{aligned}
\delta \triangleright x &>_{lpo} \delta \triangleright^* x && \text{RPO1}\\
&>_{lpo} \epsilon && \text{RPO2}
\end{aligned}$$

$$\begin{aligned}
x + y \triangleright z &>_{lpo} x + y \triangleright^* z && \text{RPO1}\\
&>_{lpo} (x + y \triangleright^* z) + (x + y \triangleright^* z) && \text{RPO2}\\
&>_{lpo} (x +^* y \triangleright z) + (x +^* y \triangleright z) && \text{RPO4, RPO5}\\
&>_{lpo} (x \triangleright z) + (y \triangleright z) && \text{RPO4, RPO5}
\end{aligned}$$

$$\begin{aligned}
a \cdot x \triangleright y &>_{lpo} a \cdot x \triangleright^* y && \text{RPO1}\\
&>_{lpo} (a \cdot x \triangleright^* y) + (a \cdot x \triangleright^* y) && \text{RPO2}\\
&>_{lpo} (a \cdot x \triangleright^* y) + \bar{a} && \text{RPO2, RPO5}\\
&>_{lpo} (a \cdot x \triangleright^* y) \cdot (a \cdot x \triangleright^* y) + \bar{a} && \text{RPO2}\\
&>_{lpo} (a \cdot^* x) \cdot (x \triangleright y) + \bar{a} && \text{RPO1, RPO3, RPO5}\\
&>_{lpo} a \cdot (x \triangleright y) + \bar{a} && \text{RPO1, RPO3}
\end{aligned}$$

$$\begin{aligned}
a \triangleright y &>_{lpo} a \triangleright^* y && \text{RPO1}\\
&>_{lpo} (a \triangleright^* y) + (a \triangleright^* y) && \text{RPO2}\\
&>_{lpo} (a \triangleright^* y) + \bar{a} && \text{RPO2, RPO5}\\
&>_{lpo} (a \triangleright^* y) \cdot (a \triangleright^* y) + \bar{a} && \text{RPO1, RPO3}\\
&>_{lpo} a \cdot y + \bar{a} && \text{RPO3, RPO5}
\end{aligned}$$

Thus, the rewrite system for PAc_ϵ is strongly normalizing. ∎

In Theorem 13.10 we showed that every PA_ϵ term has an equivalent basic term. With the definition of a basic term from Definition 13.4, we have the similar result for PAc_ϵ.

Theorem 13.19. For every PAc_ϵ term, s, there is a corresponding basic term, t, such that $PA_\epsilon \vdash s = t$. ∎

Proof. We have already shown that the subset of PAc_ϵ that corresponds to PA_ϵ can be reduced to basic terms. Thus, we only need to show that terms with the conditional operator can be reduced to basic terms.

Because the term rewriting system for PAc_ϵ is strongly normalizing by Theorem 13.18, then for every term t, there exists a finite sequence of rewritings

$$t \to t_1 \to t_2 \to \cdots \to s$$

where s is in normal form.

We use a proof by contradiction to show that s cannot contain $\triangleright$. Assume therefore, that s is in normal form and that $s = C[x \triangleright y]$.

If $x = C[u \triangleright w]$ then the argument can be applied recursively to show that $u \triangleright w$ or one of u's sub-terms can be reduced, thus contradicting that s is in normal form. Otherwise, there are five possibilities

- $x = \epsilon$: then s can be reduced by RC1.
- $x = \delta$: then s can be reduced by RC2.
- $x = u + w$: then s can be reduced by RC4.
- $x = a \cdot x'$: then s can be reduced by RC5.
- $x = a$: then s can be reduced by RC5'.

All cases contradict that s is in normal form. Thus, every PAc_ϵ term can be reduced to an equivalent basic term. ∎

13.3.4 Semantics for PAc_ϵ

The additional semantical rules for PAc_ϵ are shown in Table 13.7.

Table 13.7. Extra semantic rules for PAc_ϵ

$\dfrac{x \xrightarrow{a} x'}{x \triangleright y \xrightarrow{\bar{a}} \epsilon}$	Con1
$\dfrac{x \xrightarrow{a} x'}{x \triangleright y \xrightarrow{a} x' \triangleright y}$	Con2
$\dfrac{x \xrightarrow{a} x'}{\epsilon \triangleright x \xrightarrow{a} x'}$	Con3
$\dfrac{x\downarrow \quad y\downarrow}{x \triangleright y\downarrow}$	ConT1
$\dfrac{x \not\downarrow}{x \triangleright y\downarrow}$	ConT2

In order to prove that bisimulation is a congruence on the set of closed PAc_ϵ terms we need to introduce a generalisation of the *path* format used in the previous section. The generalisation is known as *panth* format for "predicates and *ntyft/ntyxt* hybrid format". It generalises the *path* format by allowing negative premises in the deduction rules. It is also a generalisation of the *ntyft/ntyxt* of Groote [154], which in turn along with the *path* format is a generalisation of the *tyft/tyxt* format of Groote and Vaandrager [155]. The names of these formats are derived from the format of the premises and conclusion of the deduction rules, see Verhoef [514] for an explanation.

The reference for the following material is Verhoef [514].

Definition 13.20. *(Verhoef [514])* Let $T = (\Sigma, D)$ be a term deduction system and let $D = D(T_p, T_r)$, where T_p is the set of predicate symbols and T_r is the set of relation symbols. Let I, J, K and L be index sets of arbitrary cardinality, let $s_j, t_i, u_l, v_k, t \in T(\Sigma, V)$ for all $i \in I$, $j \in J$, $k \in K$ and $l \in L$, and let $P_j, P \in T_p$ be predicate symbols for all $j \in J$, and let $R_i, R \in T_r$ be

relation symbols for all $i \in I$. A deduction rule $d \in D$ is in *panth format* if it has one of the following four forms

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{f(x_1, \ldots, x_n) R t}$$

with $f \in \Sigma$ an n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{x R t}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{P f(x_1, \ldots, x_n)}$$

with $f \in \Sigma$ and n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables or

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{P x}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables.

A term deduction system is said to be in *panth* format if all its deduction rules are in *panth* format. ∎

Before we can introduce the congruence theorem for the *panth* format we need to define some additional notions.

Definition 13.21. Let $T = (\Sigma, D)$ be a term deduction system. The formula dependency graph G of T is a labelled directed graph with the positive formulas of D as nodes. Let $PF(H)$ denote the set of all positive formulas in H and let $NF(H)$ denote all the negative formulas in H, then for all deduction rules $H/C \in D$ and for all closed substitutions σ we have the following edges in G:

- for all $h \in PF(H)$ there is an edge $\sigma(h) \xrightarrow{p} \sigma(C)$;
- for all $s \neg R \in NF(H)$ there is for all $t \in T(\Sigma)$ an edge $\sigma(sRt) \xrightarrow{n} \sigma(C)$;
- for all $\neg Ps \in NF(H)$ there is an edge $\sigma(Ps) \xrightarrow{n} \sigma(C)$.

An edge labelled with a p is called positive and an edge labelled with an n is called negative. A set of edges is called positive if all its elements are positive and negative if the edges are all negative. ∎

Definition 13.22. A term deduction system is stratifiable if there is no node in its formula dependency graph that is the start of a backward chain of edges containing an infinite negative subset. ∎

Definition 13.23. Let $T = (\Sigma, D)$ be a term deduction system and let F be a set of formulas. The variable dependency graph of F is a directed graph with the variables occurring in F as its nodes. The edge $x \to y$ is an edge of the variable dependency graph if and only if there is a positive relation $tRs \in F$ with $x \in vars(t)$ and $y \in vars(s)$.

The set F is called well-founded if any backward chain of edges in its variable dependency graph is finite. A deduction rule is called well-founded if its set of premises is so. A term deduction system is called well-founded if all its deduction rules are well-founded. ∎

We are now ready to formulate the main result of Verhoef [514].

Theorem 13.24. *(Verhoef [514]).* Let $T = (\Sigma, D)$ be a well-founded stratifiable term deduction system in *panth* format, then strong bisimulation is a congruence for all function symbols in Σ. ∎

Proof. See [514]. ∎

Lemma 13.25. Let $T = (\Sigma_{PAc_\epsilon}, D)$ be the term deduction system in Table 13.7, then strong bisimulation is a congruence on the set of closed PAc_ϵ terms. ∎

Proof. The proof relies on Theorem 13.24.

First, we must check that the term deduction system is well-founded. No variable occurs more than once in the set of premises for any of the deduction rules, so it is clear that there are no cycles in the variable dependency graph. Hence, the term deduction system is well-founded.

Next, we must show that the term deduction system is stratifiable. We use proof by contradiction. Assume the term deduction is not stratifiable. Then, there is some backward chain of edges in the formula dependency graph that contains an infinite negative subset of edges. The only negative edge in the graph is the one that stems from ConT2. Thus, there must be a cycle containing the edge $\sigma(x\downarrow) \xrightarrow{n} \sigma(x \triangleright y\downarrow)$. This cycle must also contain at least one edge originating at the node $\sigma(x \triangleright y\downarrow)$ and terminating at some node, Z, see Figure 13.24.

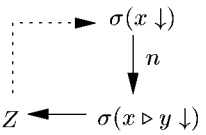

Fig. 13.24. Illustration for proof of congruence

By the definition of the formula dependency graph, the edge $\sigma(x \triangleright y\downarrow) \to Z$ can only be in the graph because there is a deduction rule with $x \triangleright y\downarrow$ as one

of its premises. However, there is no such rule, and we have a contradiction. Therefore, the term deduction system is stratifiable.

Finally, we must verify that each of the deduction rules is in *panth* format. Since any rule that is in *path* format is also in *panth* format, we only need to check the additional rules for PAc_ϵ, since the remaining rules were shown to be in *path* format in the proof for Lemma 13.15. The rule Con1 can be trivially rewritten to

$$\frac{\{x \xrightarrow{a} x'\}}{x \triangleright y \xrightarrow{\bar{a}} \epsilon}$$

which is in the first *panth* form. The rule ConT2 can similarly be rewritten to

$$\frac{\{\neg \downarrow (x)\}}{\downarrow (x \triangleright y)}$$

which is in the third *panth* form. The remaining three rules are easily shown to also be in *panth* format.

Thus, all the conditions of Theorem 13.24 are satisfied and the result follows. ∎

Theorem 13.26. The set of closed PAc_ϵ terms modulo bisimulation equivalence, notation $T(\Sigma_{PAc_\epsilon})/\sim$, is a model of PAc_ϵ. ∎

Proof. Recalling the proof for Theorem 13.16 we have to show that for each of the equations in E_{PAc_ϵ}, $PAc_\epsilon \vdash x = y$ implies the existence of a bisimulation, $\sim$, such that $x \sim y$.

We give the proof for axiom C4. Let $\sim_*$ be defined by $\{ (a \cdot x \triangleright y, a \cdot (x \triangleright y) + \bar{a}) \mid x, y \in T(\Sigma_{PAc_\epsilon}), a \in A \} \cup \{ (x, y) \mid x, y \in T(\Sigma_{PAc_\epsilon}) \}$. Clearly, $\sim_*$ is symmetric. We first check the termination condition. By ConT1 $a \cdot x \triangleright y \downarrow$, since $a \cdot x \not\downarrow$. Similarly, $a \cdot (x \triangleright y) + \bar{a} \downarrow$, since $\bar{a} \not\downarrow$ (and actually also $a \cdot (x \triangleright y) \not\downarrow$). Thus, the termination condition for bisimulation equivalence is satisfied.

Now, we check the first bisimulation condition. There are two ways $a \cdot x \triangleright y$ can evolve:

- $a \cdot x \triangleright y \xrightarrow{\bar{a}} \epsilon$: then we get $a \cdot (x \triangleright y) + \bar{a} \xrightarrow{\bar{a}} \epsilon$ and since $\epsilon \sim_* \epsilon$ by definition, the bisimulation condition is satisfied in this case.
- $a \cdot x \triangleright y \xrightarrow{a} x \triangleright y$: similarly, $a \cdot (x \triangleright y) + \bar{a} \xrightarrow{a} x \triangleright y$ and again $x \triangleright y \sim_* x \triangleright y$, so the bisimulation condition is satisfied.

The symmetric case for evolutions of $a \cdot (x \triangleright y) + \bar{a}$ is entirely analogous.

The remaining axioms can be checked with the same technique. ∎

We now come to the final result showing that the axiom system for PAc_ϵ is both sound and complete.

Theorem 13.27. The axiom system PAc_ϵ is a complete axiomatisation of the set of closed PAc_ϵ terms modulo bisimulation equivalence. ∎

Proof. Due to Theorems 13.26 and 13.19 it suffices to prove the theorem for basic terms. The proof for basic terms is given in [26]. ∎

13.4 Algebraic Semantics of Live Sequence Charts

In this section a subset of LSCs is given an algebraic semantics using the process algebra PAc_ϵ from the previous section (Sect. 13.3). The presentation here is adapted from the description of the semantics of MSC given by Mauw and Reniers [326].

13.4.1 Textual Syntax of Live Sequence Charts

We give a textual syntax for LSC. The textual syntax is used to define the semantics in the next section. The textual syntax is presented as an extended BNF (EBNF) grammar below. The nonterminals *lscid*, *msgid* and *inst name* are further unspecified identifiers. The nonterminal *cond* represents a further unspecified conditional expression.

Table 13.8. EBNF grammar for textual syntax of LSCs

⟨*chart*⟩ ::= **lsc** <lscid> ; <inst def list> **end lsc**

⟨*inst def list*⟩ ::= <inst def> <inst def list> | <>

⟨*inst def*⟩ ::= **instance** <inst name> <prechart> <body> **end instance**

⟨*prechart*⟩ ::= **prechart** <location> **end prechart**

⟨*body*⟩ ::= **body** <location> **end body**

⟨*location*⟩ ::= **hot** <event> ; <location> | **cold** <event> ; <location> | <>

⟨*event*⟩ ::= <input> | <output> | <condition> | <coregion>

⟨*input*⟩ ::= **in** <msgid> **from** <address> <mode>

⟨*output*⟩ ::= **out** <msgid > **to** <address> <mode>

⟨*condition*⟩ ::= **hot condition** <cond> | **cold condition** <cond>

⟨*coregion*⟩ ::= **concurrent** <coeventlist> **end concurrent**

⟨*coeventlist*⟩ ::= <input> <coeventlist> | <output> <coeventlist> | <>

⟨*address*⟩ ::= <inst name> | **env**

⟨*mode*⟩ ::= **sync** | **async**

We do not explain the mapping from an LSC to the textual syntax further as this is straightforward. Example 13.22 in Sect. 13.2.2 illustrates the mapping.

13.4.2 Semantics of Live Sequence Charts

In order to define the semantics of the subset of LSC, we instantiate the process algebra PAc_ϵ by specifying the set of atomic actions. We assume a set, A_o, of atomic actions representing asynchronous (*out*) and synchronous (*outs*) message output

$$\begin{aligned} A_o = &\{out(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\} \cup \\ &\{outs(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\} \end{aligned}$$

Similarly, we assume a set, A_i, of atomic actions representing asynchronous (*in*) and synchronous (*ins*) message input

$$\begin{aligned} A_i = &\{in(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\} \cup \\ &\{ins(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\} \end{aligned}$$

Conditions are also viewed as actions, so there is a set of atomic actions representing hot conditions

$$A_{hc} = \{hotcond(c) \mid c \in \mathcal{L}(\langle cond\rangle)\ \}$$

and a set of atomic actions representing cold conditions

$$A_{cc} = \{coldcond(c) \mid c \in \mathcal{L}(\langle cond\rangle)\ \}$$

The set of atomic actions, A, of the instantiated process algebra is then

$$A = A_o \cup A_i \cup A_{hc} \cup A_{cc}$$

The process algebra PAc_ϵ defined above does not place any constraints on the order of atomic events. In expressing the semantics of LSC the constraint that message input must follow the corresponding message output has to be expressed. To do this, the state operator $\lambda_{M,C}$ is introduced. It is an instance of the general state operator [27].

For $M \subseteq \mathcal{L}(\langle msgid\rangle)$, $x, y \in V$, $a \in A$, $i,j \in \mathcal{L}(\langle inst\ name\rangle)$ and $m \in \mathcal{L}(\langle msgid\rangle)$, the state operator is defined by the equations in Table 13.9. The subscript M records the message identifiers of messages that have been output, but not yet input. The subscript C records the message identifiers of those synchronous messages that have been output, but not yet input. If C is nonempty and the next event is not the corresponding input event, deadlock occurs. This ensures that no other events can come between the output and input of a synchronous message. The instantiated process algebra with $\lambda_{M,C}$ will be referred to as PA_{LSC} in the following.

The semantics of LSCs will be defined by semantic functions over the syntactical categories of the textual syntax of LSCs. If $\langle cat\rangle$ denotes a syntactical category (nonterminal) in the ENBF grammar, then $\mathcal{L}(\langle cat\rangle)$ denotes the language of text strings derivable from that syntactical category. The notation $\mathcal{P}X$ denotes the power set of the set X.

Table 13.9. Definition of state operator $\lambda_{M,C}$

$\lambda_{M,C}(\epsilon) = \epsilon$	if $M = \emptyset$
$\lambda_{M,C}(\epsilon) = \delta$	if $M \neq \emptyset$
$\lambda_{M,C}(\delta) = \delta$	
$\lambda_{M,C}(a \cdot x) = \delta$	if $a \notin A_o \cup A_i$ and $C \neq \emptyset$
$\lambda_{M,C}(a \cdot x) = a \cdot \lambda_{M,\emptyset}(x)$	if $a \notin A_o \cup A_i$ and $C = \emptyset$
$\lambda_{M,C}(out(i, env, m) \cdot x) = \delta$	if $C \neq \emptyset$
$\lambda_{M,C}(out(i, env, m) \cdot x) = out(i, env, m) \cdot \lambda_{M,\emptyset}(x)$	if $C = \emptyset$
$\lambda_{M,C}(out(i, j, m) \cdot x) = \delta$	if $m \in M$ or $C \neq \emptyset$
$\lambda_{M,C}(out(i, j, m) \cdot x) = out(i, j, m) \cdot \lambda_{M \cup \{m\},\emptyset}(x)$	if $m \notin M$ and $C = \emptyset$
$\lambda_{M,C}(outs(i, env, m) \cdot x) = \delta$	if $C \neq \emptyset$
$\lambda_{M,C}(outs(i, env, m) \cdot x) = outs(i, env, m) \cdot \lambda_{M,\emptyset}(x)$	if $C = \emptyset$
$\lambda_{M,C}(outs(i, j, m) \cdot x) = \delta$	if $m \in M$ or $C \neq \emptyset$
$\lambda_{M,C}(outs(i, j, m) \cdot x) = outs(i, j, m) \cdot \lambda_{M \cup \{m\},\{m\}}(x)$	if $m \notin M$ and $C \neq \emptyset$
$\lambda_{M,C}(in(env, j, m) \cdot x) = \delta$	if $C \neq \emptyset$
$\lambda_{M,C}(in(env, j, m) \cdot x) = in(env, j, m) \cdot \lambda_{M,\emptyset}(x)$	if $C = \emptyset$
$\lambda_{M,C}(in(i, j, m) \cdot x) = in(i, j, m) \cdot \lambda_{M \setminus \{m\},\emptyset}(x)$	if $m \in M$ and $C = \emptyset$
$\lambda_{M,C}(in(i, j, m) \cdot x) = \delta$	if $m \notin M$ or $C \neq \emptyset$
$\lambda_{M,C}(ins(env, j, m) \cdot x) = \delta$	if $C \neq \emptyset$
$\lambda_{M,C}(ins(env, j, m) \cdot x) = ins(env, j, m) \cdot \lambda_{M,\emptyset}(x)$	if $C = \emptyset$
$\lambda_{M,C}(ins(i, j, m) \cdot x) = ins(i, j, m) \cdot \lambda_{M \setminus \{m\},\emptyset}(x)$	if $m \in M$ and $C = \{m\}$
$\lambda_{M,C}(ins(i, j, m) \cdot x) = \delta$	if $m \notin M$ or $C \neq \{m\}$
$\lambda_{M,C}(x + y) = \lambda_{M,C}(x) + \lambda_{M,C}(y)$	
$\lambda_{M,C}(x \triangleright y) = \lambda_M(x) \triangleright \lambda_M(y)$	

The semantic function for LSCs,

$$S_{LSC}[\![\cdot]\!] : \mathcal{L}(\langle chart \rangle) \to T(\Sigma_{PA_{LSC}}),$$

is defined by

$$S_{LSC}[\![ch]\!] = \lambda_{\emptyset,\emptyset}\left(\left(\|_{i \in Inst_c(ch)} S_{instpc}[\![i]\!]\right) \triangleright \left(\|_{i \in Inst_c(ch)} S_{instbody}[\![i]\!]\right)\right)$$

where $Inst_c : \mathcal{L}(\langle chart \rangle) \to \mathcal{P}(\mathcal{L}(\langle inst\ def \rangle))$ is the set of instance definitions in the chart. It is defined by

$$Inst_c(\textbf{lsc}\ \langle lscid \rangle\ ;\ \langle inst\ def\ list \rangle\ \textbf{endlsc}) = Inst_{idl}(\langle inst\ def\ list \rangle)$$

where in turn $Inst_{idl} : \mathcal{L}(\langle inst\ def\ list \rangle) \to \mathcal{P}(\mathcal{L}(\langle inst\ def \rangle))$ is defined by

$$Inst_{idl}(\langle\rangle) = \emptyset$$
$$Inst_{idl}(\langle inst\ def \rangle\ \langle inst\ def\ list \rangle) = \{\langle inst\ def \rangle\} \cup Inst_{idl}(\langle inst\ def\ list \rangle)$$

The semantic function for instance precharts,

$$S_{instpc}[\![\cdot]\!] : \mathcal{L}(\langle inst\ def \rangle) \to T(\Sigma_{PA_{LSC}}),$$

is defined by

$$S_{instpc}[\![\textbf{instance } \langle inst\ name\rangle\ \langle prechart\rangle\ \langle body\rangle\ \textbf{endinstance}]\!] = S_{pre-chart}^{\langle inst\ name\rangle}[\![\langle prechart\rangle]\!]$$

The semantic function for instance bodies,

$$S_{instbody}[\![\cdot]\!] : \mathcal{L}(\langle inst\ def\rangle) \to T(\Sigma_{PA_{LSC}}),$$

is defined by

$$S_{instbody}[\![\textbf{instance } \langle inst\ name\rangle\ \langle prechart\rangle\ \langle body\rangle\ \textbf{endinstance}]\!] = S_{body}^{\langle inst\ name\rangle}[\![\langle body\rangle]\!]$$

For $iid \in \mathcal{L}(\langle inst\ name\rangle$ the semantic function for precharts,

$$S_{body}^{iid}[\![\cdot]\!] : \mathcal{L}(\langle prechart\rangle) \to T(\Sigma_{PA_{LSC}}),$$

is defined by

$$S_{prechart}^{iid}[\![\textbf{prechart } \langle location\rangle\ \textbf{endprechart}]\!] = S_{location}^{iid}[\![\langle location\rangle]\!]$$

For $iid \in \mathcal{L}(\langle inst\ name\rangle$ the semantic function for instance bodies,

$$S_{body}^{iid}[\![\cdot]\!] : \mathcal{L}(\langle body\rangle) \to T(\Sigma_{PA_{LSC}}),$$

is defined by

$$S_{body}^{iid}[\![\textbf{body } \langle location\rangle\ \textbf{endbody}]\!] = S_{location}^{iid}[\![\langle location\rangle]\!]$$

For $iid \in \mathcal{L}(\langle inst\ name\rangle$ the semantic function for event lists,

$$S_{location}^{iid}[\![\cdot]\!] : \mathcal{L}(\langle location\rangle) \to T(\Sigma_{PA_{LSC}}),$$

is defined by:

$$S_{location}^{iid}[\![\langle\rangle]\!] = \epsilon$$
$$S_{location}^{iid}[\![\textbf{hot } \langle event\rangle\ ;\ \langle location\rangle]\!] = S_{event}^{iid}[\![\langle event\rangle]\!] \cdot S_{location}^{iid}[\![\langle location\rangle]\!]$$
$$S_{location}^{iid}[\![\textbf{cold } \langle event\rangle\ ;\ \langle location\rangle]\!] = \epsilon + \left(S_{event}^{iid}[\![\langle event\rangle]\!] \cdot S_{location}^{iid}[\![\langle location\rangle]\!]\right)$$

For $iid \in \mathcal{L}(\langle inst\ name\rangle$ the semantic function for events,

$$S_{event}^{iid}[\![\cdot]\!] : \mathcal{L}(\langle event\rangle) \to T(\Sigma_{PA_{LSC}}),$$

is defined by:

$$S^{iid}_{event}[\![\textbf{out } \langle msgid \rangle \textbf{ to } \langle address \rangle \textbf{ async}]\!] = out(iid, \langle address \rangle, \langle msgid \rangle)$$
$$S^{iid}_{event}[\![\textbf{out } \langle msgid \rangle \textbf{ to } \langle address \rangle \textbf{ sync}]\!] = outs(iid, \langle address \rangle, \langle msgid \rangle)$$
$$S^{iid}_{event}[\![\textbf{in } \langle msgid \rangle \textbf{ from } \langle address \rangle \textbf{ async}]\!] = in(\langle address \rangle, iid, \langle msgid \rangle)$$
$$S^{iid}_{event}[\![\textbf{in } \langle msgid \rangle \textbf{ from } \langle address \rangle \textbf{ sync}]\!] = ins(\langle address \rangle, iid, \langle msgid \rangle)$$
$$S^{iid}_{event}[\![\textbf{hotcondition } \langle cond \rangle]\!] = hotcond(\langle cond \rangle)$$
$$S^{iid}_{event}[\![\textbf{coldcondition } \langle cond \rangle]\!] = coldcond(\langle cond \rangle)$$
$$S^{iid}_{event}[\![\textbf{concurrent } \langle coeventlist \rangle \textbf{ endconcurrent}]\!] = \|_{e \in CoEvents(\langle coeventlist \rangle)} S^{iid}_{event}[\![e]\!]$$

where $CoEvents : \mathcal{L}(\langle eventlist \rangle) \to \mathcal{P}(\mathcal{L}(\langle event \rangle))$ is defined by:

$$CoEvents(\langle\rangle) = \emptyset$$
$$CoEvents(\langle event \rangle \ \langle eventlist \rangle) = \{\langle event \rangle\} \cup CoEvents(\langle eventlist \rangle)$$

13.4.3 The Live Sequence Chart Example, II

Example 13.23 *The Live Sequence Chart, Part II:* We end this section with an example that concludes Example 13.22 of Sect. 13.2.2. The example illustrates the process of deriving a PA_{LSC} term from the LSC diagram of Sect. 13.2.2. We derive the PA_{LSC} term from the textual syntax by using the semantic function for LSCs. Let the chart be denoted by ch, then the semantics of ch is given by the PA_{LSC} term below.

$$\begin{aligned}
&S_{LSC}[\![ch]\!] = \\
&\quad \lambda_{\emptyset,\emptyset}((hotcond(cond_1) \cdot out(A,B,m_1) \parallel \\
&\qquad hotcond(cond_1) \cdot in(A,B,m_1)) \\
&\qquad \triangleright \\
&\qquad (out(A,B,m_2) \cdot coldcond(cond_2) \cdot outs(A,B,m_4) \\
&\qquad \parallel \\
&\qquad (in(A,B,m_2) \parallel in(C,A,m_3)) \cdot coldcond(cond_2) \cdot \\
&\qquad ins(A,B,m_4) \cdot out(B,C,m_3) \\
&\qquad \parallel \\
&\qquad out(C,B,m_3) \cdot (\epsilon + in(B,C,m_5) \cdot (\epsilon + out(C,env,m_6)))))
\end{aligned}$$

■

13.5 Relating Message Charts to RSL

In this section, as well as in Sect. 14.7, we briefly review a number of ways of integrating different specification notations. We then define a subset of RSL and give an operational semantics based on the semantics for Timed RSL as defined by George and Xia [132] (see Sect. 15.4). We extend the semantic rules with behaviour annotations capturing the communication behaviour of

the RSL expression. Utilizing these behaviours, we define three satisfaction relations: one relating a universal LSC to an RSL specification, one relating an existential LSC to an RSL specification and, in Sect. 14.7, one relating a statechart to an RSL specification.

13.5.1 Types of Integration

Haxthausen [203] identifies three approaches to integrating different specification techniques:

- the *unifying, wide-spectrum* approach
- the *family* approach
- the *linking* approach

The wide-spectrum approach provides a complete semantical integration of the techniques. This was the approach adopted in the development of RSL. The advantage of this approach is that the same language is used throughout the development process. The disadvantage is that this approach results in a complicated semantics.

The idea in the family approach is to define a reasonably expressive base language and then integrate other techniques by defining extension languages. The semantics of the extension languages are required to be consistent with the semantics of the base language. This approach is used in the CoFI [371] project, for which the base language is called CASL [40]. The advantage of the family approach is that the semantics is "only as complicated as it needs to be", in the sense that for a particular project, one uses the smallest language in the family that has the required facilities.

In the previous two approaches a new semantics that subsumes the semantics of the individual techniques is developed. In contrast, in the linking approach the individual semantics are preserved, and the integration instead takes the form of a formal relation between the individual semantics. This approach is particularly suited for specification techniques that are fundamentally different.

There is also a fourth approach to integration, namely what we call the combination approach. In this approach one notation is embedded in the other to extend its expressiveness. An example is the coloured Petri nets, which are the result of the combination of classical Petri nets with an ML-like language [238, 275] used for inscriptions on arcs and type definitions. Other examples are the combinations of statecharts with CASL and statecharts with Z mentioned in the introduction.

We believe that of the four approaches described, the linking approach is most suited for our purpose. By using this approach we do not have to "massage" the familiar semantics of the individual techniques into a new framework. Additionally, all the tools (proof system, syntax checkers, code generators) developed for RSL are immediately available in the integrated method.

In the rest of this chapter we therefore present how to link LSCs with RSL. In the next chapter we will explain how to link statecharts with RSL.

13.5.2 An RSL Subset

Syntax

The subset of RSL defined below is almost the same as the subset defined by George and Xia [132] for Timed RSL. We omit the **wait** construct and use the standard input and output operators from RSL rather than the corresponding operators in timed RSL (TRSL, [132]). Also, we exclude the special notation for recursive functions. For use in establishing the relation to LSCs and statecharts, we annotate the input and output operators with a message identifier. Similarly, the parallel and interlocking operators are annotated with two process identifiers.

We assume familiarity with RSL and therefore skip an informal description of the operators and constructs of the RSL subset.

The syntactic categories are

- expressions denoted by E
- variables denoted by x
- identifiers denoted by id
- channels denoted by c
- reals denoted by r
- types denoted by τ
- value definitions denoted by V
- message identifiers denoted by $msgid$
- process identifiers denoted by n

The grammar of the subset of RSL is given below.

V ::= id : τ | id : τ, V

E ::= () | **true** | **false** | r | id | x | **skip** | **stop** | **chaos**
| x := E | **if** E **then** E **else** E | **let** id = E **in** E | c?$_{msgid}$ | c!$_{msgid}$E
| E $\prod$ E | E $[]$ E | E $_n\|\|_n$ E | E $_n\|\|_n$ E | E ; E
| λ id : τ • E | E E

When in the following we refer to an RSL expression, we mean an expression within the subset of RSL defined here.

Operational Semantics with Communication Behaviour

Before presenting the rules of the operational semantics a number of definitions are needed. A store s is a finite map from variables (x) to values (v): $s = [x \mapsto v, \ldots]$. An environment ρ is a finite map from identifiers (id) to values (v) : $\rho = [id \mapsto v, \ldots]$. A closure is a pair consisting of a lambda expression (λ id : τ • E) and an environment (ρ): $[\![\lambda \text{ id} : \tau \bullet \text{E}, \rho]\!]$.

Compared to George and Xia [132], we modify the notion of a configuration to a triple $< E, s, n >$, where E is an expression, s is a store and n is a process identifier. Moreover, we augment configurations of the form α *op* s *op* β for *op* $= \|, \nparallel$ to include three process identifiers, i.e., α *op* (s, n, n_1, n_2) *op* β, where n_1 is the identifier of the process represented by the configuration α, while n_2 is the identifier of the process represented by β.

Inspired by Haxthausen and Xia [204], the rules of the operational semantics are extended to include communication behaviour in the form of a PA_{LSC} term. The transition relation has the form

$$\rho \vdash \alpha_{\textbf{with } \phi} \xrightarrow{e} \alpha'_{\textbf{with } \phi'}$$

where ρ is the environment, α and α' are configurations, ϕ and ϕ' are behaviours and e is an event. The intuition is that the configuration α with the behaviour ϕ can evolve to the configuration α' with behaviour ϕ' by performing the event e.

There are two types of events, silent events and communication events. The silent event, ϵ, denotes an internal change that is not externally visible. Communication events are either input events of the form $c?_{msgid}$ or output events of the form $c!_{msgid}E$. The symbol $\diamond$ is used to denote any event, i.e., a situation where the transition is the same for a silent event and for a communication event.

The only operational rules that change the communication behaviour are the rules for input, output, communication across a parallel or interlocking combinator and merging of two parallel processes. In all other rules, the communication behaviour is preserved.

The process identifier, n, stored in a configuration is used to name processes in PA_{LSC} events. This information is needed to identify the sender and recipient in message input and message output events in the behaviours.

The rules for the parallel and interlocking combinators apply the function *merge* that merges the stores on either side of a parallel composition. It is defined in RSL notation by:

value
merge(s,s',s'') ≡ s'†[x↦s''(x) | x ∈ **dom**(s'')∩ **dom**(s)•s(x)≠s''(x)]

In the rules below we use a notation of the form:

$$\frac{C}{\rho \vdash \begin{array}{l} C_2 \\ C_3 \end{array}}$$

as a shorthand for the two rules:

$$\frac{C}{\rho \vdash C_2}$$

and:

$$\frac{C}{\rho \vdash C_3}.$$

Also, for rules without premises, i.e., axioms, we write the symbol □ above the line.

Tables 13.10–13.23 each contain one rule. They are:

- Basic Expressions Table 13.10
- Configuration Fork Table 13.11
- Look Up Table 13.12
- Sequencing Table 13.13
- Assignment Table 13.14
- Input Table 13.15
- Output Table 13.16
- Internal Choice Table 13.17
- External Choice Table 13.18
- Parallel Combinator Table 13.19
- Interlocking Combinator Table 13.20
- Function Table 13.21
- Let Expression Table 13.22
- If Expression Table 13.23

Table 13.10. Basic expressions

$$\frac{\square}{\rho \vdash < \mathbf{skip}, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < (), s, n >_{\mathbf{with}\ \phi}}$$

$$\frac{\square}{\rho \vdash < \mathbf{chaos}, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < \mathbf{chaos}, s, n >_{\mathbf{with}\ \phi}}$$

Table 13.11. Configuration fork

$$\frac{\square}{\rho \vdash < E_1\ op\ E_2, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < E_1, s, n >_{\mathbf{with}\ \phi}\ \ op\ \ < E_2, s, n >_{\mathbf{with}\ \phi}}$$

where $op \in \{[\!]\, , \sqcap\}$

Table 13.12. Look up

$$\frac{}{\rho \dagger [id \mapsto v] \vdash < id, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < v, s, n >_{\mathbf{with}\ \phi}}\ \square$$

$$\frac{}{\rho \vdash < id, s \dagger [id \mapsto v], n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < v, s \dagger [id \mapsto v], n >_{\mathbf{with}\ \phi}}\ \square$$

Table 13.13. Sequencing

$$\frac{}{\rho \vdash < E_1; E_2, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} (< E_1, s, n >; E_2)_{\mathbf{with}\ \phi}}\ \square$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\rho \vdash (\alpha; E)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} (\alpha'; E)_{\mathbf{with}\ \phi'}}$$

$$\frac{}{\rho \vdash (< v, s, n >; E)_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < E, s, n >_{\mathbf{with}\ \phi}}\ \square$$

Table 13.14. Assignment

$$\frac{}{\rho \vdash < x := E, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} (x :=< E, s, n >)_{\mathbf{with}\ \phi}}\ \square$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\rho \vdash (x := \alpha)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} (x := \alpha')_{\mathbf{with}\ \phi'}}$$

$$\frac{}{\rho \vdash < v, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < (), s \dagger [x \mapsto v], n >_{\mathbf{with}\ \phi}}\ \square$$

13.5.3 Relating Live Sequence Charts to RSL

Syntactical Restrictions

There are a number of problematic issues with conditions in LSCs. For that reason we choose to omit hot and cold conditions when relating an RSL specification to an LSC. This is done by removing all condition events from the PA_{LSC} term prior to checking satisfaction.

Table 13.15. Input

$$\frac{}{\rho \vdash < c?_{msgid}, s, n >_{\textbf{with } \phi} \xrightarrow{c?_{msgid} v} < v, s, n >_{\textbf{with } \phi \cdot ins(env,n,msgid)}} \square$$

Table 13.16. Output

$$\frac{}{\rho \vdash < c!_{msgid} E, s, n >_{\textbf{with } \phi} \xrightarrow{\epsilon} (c!_{msgid} < E, s, n >)_{\textbf{with } \phi}} \square$$

$$\frac{\rho \vdash \alpha_{\textbf{with } \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with } \phi'}}{\rho \vdash (c!_{msgid}\alpha)_{\textbf{with } \phi} \xrightarrow{\diamond} (c!_{msgid}\alpha')_{\textbf{with } \phi'}}$$

$$\frac{}{\rho \vdash (c!_{msgid} < v, s, n >)_{\textbf{with } \phi} \xrightarrow{c!_{msgid} v} < (), s, n >_{\textbf{with } \phi \cdot outs(n,env,msgid)}} \square$$

Table 13.17. Internal choice

$$\frac{}{\rho \vdash (\alpha \prod \beta)_{\textbf{with } \phi} \xrightarrow{\epsilon} \alpha_{\textbf{with } \phi} \quad \xrightarrow{\epsilon} \beta_{\textbf{with } \phi}} \square$$

Since RSL only supports synchronous communication on channels, we restrict the relation to cover synchronous messages only. More specifically, if an LSC contains asynchronous messages, no RSL specification can satisfy it.

Satisfaction Relation

Before we can define what it means for an RSL expression to satisfy an LSC, we introduce some auxiliary notions. In most cases we do not want an LSC to constrain all parts of an RSL specification. Typically, we only want to constrain the sequence of a limited number of messages. For this reason we label each LSC with the set of events it constrains. We allow this set to contain events that are not mentioned in the chart. For an LSC *ch* this set is denoted $\mathcal{C}_{ch}$.

Below we need an event extraction function that yields the set of those event identifiers that occur in the PA_{LSC} term for an LSC. The event extraction function, $events : T(\Sigma_{PAc_\epsilon}) \to \mathcal{P}Event$, is defined as

Table 13.18. External choice

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{a} \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{l}\rho \vdash \alpha_{\mathbf{with}\ \phi} \,[]\, \beta_{\mathbf{with}\ \varphi} \xrightarrow{a} \alpha'_{\mathbf{with}\ \phi'} \\ \quad\ \beta_{\mathbf{with}\ \varphi} \,[]\, \alpha_{\mathbf{with}\ \phi} \xrightarrow{a} \alpha'_{\mathbf{with}\ \phi'}\end{array}}$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{l}\rho \vdash \alpha_{\mathbf{with}\ \phi} \,[]\, \beta_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} \alpha'_{\mathbf{with}\ \phi'} \,[]\, \beta_{\mathbf{with}\ \varphi} \\ \quad\ \beta_{\mathbf{with}\ \varphi} \,[]\, \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} \beta_{\mathbf{with}\ \varphi} \,[]\, \alpha'_{\mathbf{with}\ \phi'}\end{array}}$$

$$\frac{\square}{\begin{array}{l}\rho \vdash < v, s, n >_{\mathbf{with}\ \phi} \,[]\, \alpha_{\mathbf{with}\ \phi'} \xrightarrow{\epsilon} < v, s, n >_{\mathbf{with}\ \phi} \\ \quad\ \alpha_{\mathbf{with}\ \phi'} \,[]\, < v, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < v, s, n >_{\mathbf{with}\ \phi}\end{array}}$$

$$\begin{aligned}
&events(\epsilon) = \emptyset \\
&events(in(n_1, n_2, m)) = \{m\} \\
&events(out(n_1, n_2, m)) = \{m\} \\
&events(ins(n_1, n_2, m)) = \{m\} \\
&events(outs(n_1, n_2, m)) = \{m\} \\
&events(hotcondition(cond)) = \emptyset \\
&events(coldcondition(cond)) = \emptyset \\
&events(X \cdot Y) = events(X) \cup events(Y) \\
&events(X + Y) = events(X) \cup events(Y) \\
&events(X \parallel Y) = events(X) \cup events(Y) \\
&events(X \triangleright Y) = events(X) \cup events(Y) \\
&remcond(\epsilon) = \epsilon
\end{aligned}$$

As explained above, we do not check LSC conditions when making the relation to RSL. The function removing conditions, $remcond : T(\Sigma_{PAc_\epsilon}) \to T(\Sigma_{PAc_\epsilon})$, is defined as:

$$\begin{aligned}
&remcond(in(n_1, n_2, m)) = in(n_1, n_2, m) \\
&remcond(out(n_1, n_2, m)) = out(n_1, n_2, m) \\
&remcond(ins(n_1, n_2, m)) = in(n_1, n_2, m) \\
&remcond(outs(n_1, n_2, m)) = out(n_1, n_2, m)
\end{aligned}$$

Table 13.19. Parallel combinator

$$\frac{}{\rho \vdash < E_1\ {}_{n_1}\|_{n_2}\ E_2, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < E_1, s, n_1 >_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| < E_1, s, n_2 >_{\mathbf{with}\ \phi}} \ \square$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{c!_{msgid}v} \alpha'_{\mathbf{with}\ \phi'} \qquad \rho \vdash \beta_{\mathbf{with}\ \varphi} \xrightarrow{c?_{msgid}v} \beta'_{\mathbf{with}\ \varphi'}}{\begin{array}{l}\rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| \beta_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} \alpha'_{\mathbf{with}\ \phi \cdot out(n_1, n_2, id)} \| (s, n, n_1, n_2) \\ \qquad\qquad \| \beta'_{\mathbf{with}\ \varphi \cdot in(n_1, n_2, msgid)} \\ \beta_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} \beta'_{\mathbf{with}\ \varphi \cdot in(n_2, n_1, id)} \| (s, n, n_1, n_2) \\ \qquad\qquad \| \alpha'_{\mathbf{with}\ \phi \cdot out(n_2, n_1, msgid)}\end{array}}$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{l}\rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| \beta_{\mathbf{with}\ \varphi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'} \| (s, n, n_1, n_2) \| \beta_{\mathbf{with}\ \varphi} \\ \beta_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \beta_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha'_{\mathbf{with}\ \phi'}\end{array}}$$

$$\frac{}{\rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| < v, s', n_2 >_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \\ < v, s', n_2 >_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi}} \ \square$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{l}\rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'} \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \\ s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha'_{\mathbf{with}\ \phi'}\end{array}}$$

$$\frac{}{\rho \vdash \begin{array}{l} < v, s'', n_1 >_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\mathbf{with}\ \phi \| \varphi} \\ s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| < v, s'', n_1 >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\mathbf{with}\ \phi \| \varphi} \end{array}} \ \square$$

$$\begin{aligned}
&remcond(hotcondition(cond)) = \epsilon \\
&remcond(coldcondition(cond)) = \epsilon \\
&remcond(X \cdot Y) = remcond(X) \cdot remcond(Y) \\
&remcond(X + Y) = remcond(X) + remcond(Y) \\
&remcond(X \parallel Y) = remcond(X) \parallel remcond(Y) \\
&remcond(X \triangleright Y) = remcond(X) \triangleright remcond(Y)
\end{aligned}$$

Definition. A PA_{LSC} term, x, can *simulate* a PAc_ϵ term, y, notation $x \succeq y$, if

$$y \downarrow \Rightarrow x \downarrow \ \wedge \forall y' : y \xrightarrow{a} y' \Rightarrow \exists x' : x \xrightarrow{a} x' \wedge x' \succeq y'$$

■

Table 13.20. Interlocking combinator

$$\frac{}{\rho \vdash < E_1\ {}_{n_1}\|_{n_2} E_2, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < E_1, s, n_1 >_{\mathbf{with}\ \phi} \ \| \ (s, n, n_1, n_2) \ \| \ < E_1, s, n_2 >_{\mathbf{with}\ \phi}} \ \square$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{c!_{msgid}{}^{v}} \alpha'_{\mathbf{with}\ \phi'} \qquad \rho \vdash \beta_{\mathbf{with}\ \varphi} \xrightarrow{c?_{msgid}{}^{v}} \beta'_{\mathbf{with}\ \varphi'}}{\begin{array}{l} \rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| \beta_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} \alpha'_{\mathbf{with}\ \phi \cdot out(n_1, n_2, id)} \| (s, n, n_1, n_2) \\ \qquad\qquad \| \beta'_{\mathbf{with}\ \varphi \cdot in(n_1, n_2, msgid)} \\ \beta_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} \beta'_{\mathbf{with}\ \varphi \cdot in(n_2, n_1, id)} \| (s, n, n_1, n_2) \\ \qquad\qquad \| \alpha'_{\mathbf{with}\ \phi \cdot out(n_2, n_1, msgid)} \end{array}}$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{l} \rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| \beta_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} \alpha'_{\mathbf{with}\ \phi'} \| (s, n, n_1, n_2) \| \beta_{\mathbf{with}\ \varphi} \\ \beta_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} \beta_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha'_{\mathbf{with}\ \phi'} \end{array}}$$

$$\frac{}{\begin{array}{l} \rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| \ < v, s', n_2 >_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \\ \quad < v, s', n_2 >_{\mathbf{with}\ \varphi} \ \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \end{array}} \ \square$$

$$\frac{\rho \vdash \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{l} \rho \vdash \alpha_{\mathbf{with}\ \phi} \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'} \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \\ s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \alpha'_{\mathbf{with}\ \phi'} \end{array}}$$

$$\frac{}{\begin{array}{l} \rho \vdash \ < v, s'', n_1 >_{\mathbf{with}\ \phi} \ \| (s, n, n_1, n_2) \| s'_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\mathbf{with}\ \phi \| \varphi} \\ \quad s'_{\mathbf{with}\ \varphi} \| (s, n, n_1, n_2) \| \ < v, s'', n_1 >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\mathbf{with}\ \phi \| \varphi} \end{array}} \ \square$$

Definition. A PA_{LSC} formula *cbh* is called a *communication behaviour* of an RSL expression E wrt. an initial store s_0, if and only if there exists a configuration α, such that

$$[\,] \vdash < E, s_0, n >_{\mathbf{with}\ \epsilon} \quad (\xrightarrow{\diamond})^* \quad \alpha_{\mathbf{with}\ cbh}$$

where $(\xrightarrow{\diamond})^*$ denotes the transitive closure of the transition relation. If α is of the form $< v, s, n >$, where v is a value literal or a lambda expression, *cbh* is called a *terminated behaviour*. ∎

We are now ready to define the satisfaction relations for universal and existential LSCs.

Table 13.21. Function

$$\frac{}{\rho \vdash < E_1\ E_2, s, n >_{\textbf{with}\ \phi} \xrightarrow{\epsilon} (< E_1, s, n > E_2)_{\textbf{with}\ \phi}}\ \square$$

$$\frac{\rho \vdash \alpha_{\textbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with}\ \phi'}}{\rho \vdash (\alpha\ E)_{\textbf{with}\ \phi} \xrightarrow{\diamond} (\alpha'\ E)_{\textbf{with}\ \phi'}}$$

$$\frac{}{\rho \vdash < \lambda\, id : \tau \bullet E, s, n >_{\textbf{with}\ \phi} \xrightarrow{\epsilon} < [\![\lambda\, id : \tau \bullet E, \rho]\!], s, n >_{\textbf{with}\ \phi}}\ \square$$

$$\frac{}{\rho \vdash (< [\![\lambda\, id : \tau \bullet E_1, \rho_1]\!], s, n >\ E_2)_{\textbf{with}\ \phi} \xrightarrow{\epsilon} ([\![\lambda\, id : \tau \bullet E_1, \rho_1]\!] < E_2, s, n >)_{\textbf{with}\ \phi}}\ \square$$

$$\frac{\rho \vdash \alpha_{\textbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with}\ \phi'}}{\rho \vdash ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ \alpha)_{\textbf{with}\ \phi} \xrightarrow{\diamond} ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ \alpha')_{\textbf{with}\ \phi'}}$$

$$\frac{}{\rho \vdash ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ < v, s, n >)_{\textbf{with}\ \phi} \xrightarrow{\diamond} ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ v)_{\textbf{with}\ \phi}}\ \square$$

$$\frac{\rho_1 \dagger [id \mapsto v] \vdash \alpha_{\textbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with}\ \phi'}}{\rho \vdash ([\![\lambda\, id : \tau \bullet \alpha, \rho_1]\!]\ v)_{\textbf{with}\ \phi} \xrightarrow{\diamond} ([\![\lambda\, id : \tau \bullet \alpha', \rho_1]\!]\ v)_{\textbf{with}\ \phi'}}$$

$$\frac{\rho_1 \dagger [id \mapsto v] \vdash \alpha_{\textbf{with}\ \phi} \xrightarrow{\diamond} < v', s, n >_{\textbf{with}\ \phi'}}{\rho \vdash ([\![\lambda\, id : \tau \bullet \alpha, \rho_1]\!]\ v)_{\textbf{with}\ \phi} \xrightarrow{\diamond} < v', s, n >_{\textbf{with}\ \phi'}}$$

Definition. (Satisfaction for universal LSC) An RSL expression E *satisfies* a universal LSC, ch, if for any initial store, s_0, for any terminated behaviour, cbh, of E there exists a PA_{LSC} term ϕ_{prefix} and a PA_{LSC} term ϕ_{suffix}, such that

$$\begin{aligned} events(\phi_{\text{prefix}}) \cap \mathcal{C}_{ch} &= \emptyset \\ events(\phi_{\text{suffix}}) \cap \mathcal{C}_{ch} &= \emptyset \end{aligned}$$

and

$$\phi_{\text{prefix}} \cdot remcond(S_{LSC}[\![ch]\!]) \cdot \phi_{\text{suffix}} \succeq cbh$$

■

Definition. (Satisfaction for existential LSC) An RSL expression E *satisfies* an existential LSC, ch, if for any initial store, s_0, there exists a terminated behaviour, cbh, of E, a PA_{LSC} term ϕ_{prefix} and a PA_{LSC} term ϕ_{suffix}, such that

Table 13.22. Let expression

$$\frac{\square}{\rho \vdash < \textbf{let } id = E_1 \textbf{ in } E_2, s, n >_{\textbf{with } \phi} \xrightarrow{\epsilon} (\textbf{let } id =< E_1, s, > \textbf{ in } E_2)_{\textbf{with } \phi}}$$

$$\frac{\rho \vdash \alpha_{\textbf{with } \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with } \phi'}}{\rho \vdash (\textbf{let } id = \alpha \textbf{ in } E)_{\textbf{with } \phi} \xrightarrow{\diamond} (\textbf{let } id = \alpha' \textbf{ in } E)_{\textbf{with } \phi'}}$$

$$\frac{\square}{\rho \vdash (\textbf{let } id =< v, s, n > \textbf{ in } E)_{\textbf{with } \phi} \xrightarrow{\epsilon} < E[v/id], s, n >_{\textbf{with } \phi}}$$

Table 13.23. If expression

$$\frac{\square}{\rho \vdash < \textbf{if } E \textbf{ then } E_1 \textbf{ else } E_2, s, n >_{\textbf{with } \phi} \xrightarrow{\epsilon} (\textbf{if } < E, s, n > \textbf{ then } E_1 \textbf{ else } E_2)_{\textbf{with } \phi}}$$

$$\frac{\rho \vdash \alpha_{\textbf{with } \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with } \phi'}}{\rho \vdash (\textbf{if } \alpha \textbf{ then } E_1 \textbf{ else } E_2)_{\textbf{with } \phi} \xrightarrow{\diamond} (\textbf{if } \alpha' \textbf{ then } E_1 \textbf{ else } E_2)_{\textbf{with } \phi'}}$$

$$\frac{\square}{\rho \vdash (\textbf{if } < \textbf{true}, s, n > \textbf{ then } E_1 \textbf{ else } E_2)_{\textbf{with } \phi} \xrightarrow{\epsilon} < E_1, s, n >_{\textbf{with } \phi}}$$

$$\frac{\square}{\rho \vdash (\textbf{if } < \textbf{false}, s, n > \textbf{ then } E_1 \textbf{ else } E_2)_{\textbf{with } \phi} \xrightarrow{\epsilon} < E_2, s, n >_{\textbf{with } \phi}}$$

$$events(\phi_{\text{prefix}}) \cap \mathcal{C}_{ch} = \emptyset$$
$$events(\phi_{\text{suffix}}) \cap \mathcal{C}_{ch} = \emptyset$$

and

$$\phi_{\text{prefix}} \cdot remcond(S_{LSC}[\![ch]\!]) \cdot \phi_{\text{suffix}} \succeq cbh$$

■

13.5.4 Checking Satisfaction

The satisfaction criteria defined in Definition 13.5.3 require checking that all behaviours of the RSL expression can be simulated by the semantics of the corresponding chart. In some situations the RSL expressions may have infinitely many behaviours, so in that case, this simple form of checking is not possible.

13.5.5 Tool Support

Actually checking an RSL specification against a behavioural specification in the form of LSCs can be very tedious. For that reason, the methods defined above are of limited applicability without tool support. Tools should be developed to extract the semantic terms from LSCs and RSL specifications and for checking the satisfaction relations. It would also be convenient to have a way of translating an LSC into a skeleton RSL specification. An automatic conversion would force the software engineer to use one particular style.

13.6 Communicating Transaction Processes (CTP)

Section 13.6 is the joint work of Yang Shaofa and Dines Bjørner. Yang provided the Dining Philosophers example, Sect. 13.6.3, and the formalisation, Sect. 13.6.4.

We refer to the published paper [439]. CTPs are formed by a relatively simple and elegant composition of Petri net places and sets of message sequence charts.

13.6.1 Intuition

CTPs are motivated by considering first a Petri net such as the one depicted in the upper half of Fig. 13.25. The conditions (or places) are labelled $S_{P_{1_1}}, S_{P_{1_2}}, S_{P_{1_3}}, S_{P_{2_1}}, S_{P_{2_2}}, S_{P_{3_1}}$ and $S_{P_{3_2}}$. The events (or transitions) are labelled T_1, T_2 and T_3. Our labelling of places reflects a pragmatic desire to group three of these $(S_{P_{1_1}}, S_{P_{1_2}}, S_{P_{1_3}})$ into what we may then call control states of a process P_1, two of these $(S_{P_{2_1}}, S_{P_{2_2}})$ into control states of process P_2 and the remaining two $(S_{P_{2_1}}, S_{P_{2_2}})$ into control states of process P_3.

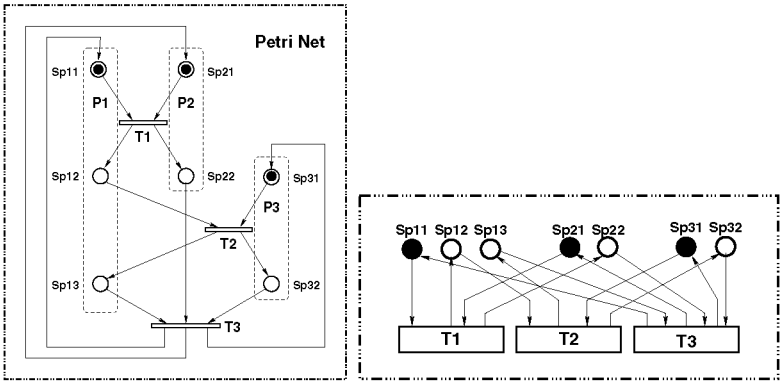

Fig. 13.25. Left: a Petri net. Right: a concrete CTP diagram

Secondly we consider each event as a message sequence chart. T_1 has two instances corresponding to processes P_1 and P_2. For that (and the below implied) message sequence chart(s) messages are being specified for communication between these instances and internal actions are being specified for execution. The firing of event T_1 shall thus correspond to the execution of this message sequence chart. T_2 has two instances corresponding to processes P_2 and P_3 and T_3 has three instances corresponding to processes P_1, P_2 and P_3.

As for condition event Petri nets, tokens are placed in exactly one of the control states for each process. Enabling and firing take place as for condition event Petri nets. Transfer of tokens from input places to output places shall take place in two steps. First when invoking the transition message sequence chart where tokens are removed from enabling input places, and then when all instances of the invoked message sequence chart have been completed (where tokens are placed at designated output places).

Thirdly we consider each event as a set of one or more message sequence charts with all message sequence charts of any given event involving the same processes. In doing so, we refine each event into a transaction schema. There is now the question as to which of the message sequence charts is to be selected. That question is clarified by the fourth step motivating CTPs.

Fourthly we predicate the selection of which message sequence charts are to be selected once a transaction schema is fired by equipping each of the message sequence charts with a guard, that is, a proposition. Associated with each process there is a set of local variables that can be, and usually are updated by the internal actions of the instances. The propositions are the conjunctions of one proposition for each of the instances, i.e., processes. A message sequence chart of a transaction schema is enabled if its guard evaluates to true. If two or more message sequence charts are enabled one is nondeterministically (internal choice) selected. A transaction schema is enabled if its input places are marked and at least one of the message sequence charts in this transaction schema is enabled. If a transaction schema has no message sequence charts enabled, then we will not enter this transaction schema.

We are now ready to introduce CTPs properly.

13.6.2 Narration of CTPs

CTP Diagrams

Consider Fig. 13.26. It is a generalisation of the right part of Fig. 13.25 which itself is just a reformatting of the left part of Fig. 13.25.

A CTP diagram consists of **an indexed set of sets of process (control) states,** an indexed set of transaction schemas, an indexed set of sets of process variables, and a "wiring" connecting control states via transaction schemas to control states. (The wiring of Fig. 13.26 is shown by pairs of opposite directed arrows.)

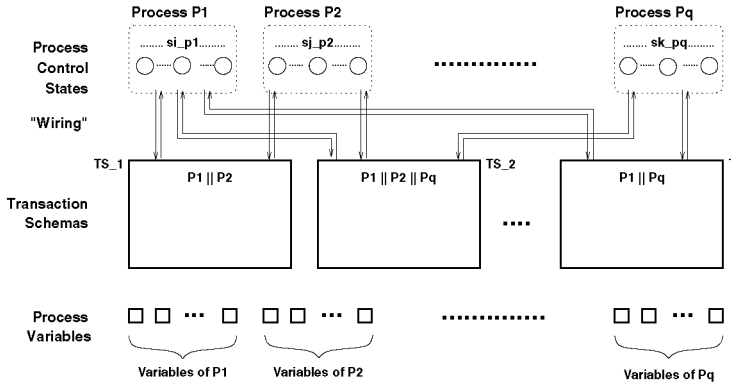

Fig. 13.26. A schematic CTP diagram

CTP Processes

Figure 13.26 suggests a notion of processes, here named p_1, p_2, ..., p_q (in Fig. 13.26 $P1$, $P2$, ..., Pq). It also suggests a number of transaction schemas, here named TS_1, TS_2, ..., TS_s. The figure then suggests that the processes have the following control states:

- $p_1 : \{s^1_{p_1}, s^2_{p_1}, \ldots, s^{m_1}_{p_1}\}$ in Fig. 13.26: si_p1,
- $p_2 : \{s^1_{p_2}, s^2_{p_2}, \ldots, s^{m_2}_{p_2}\}$ in Fig. 13.26: sj_p2,
- ...
- $p_q : \{s^1_{p_q}, s^2_{p_q}, \ldots, s^{m_q}_{p_q}\}$ in Fig. 13.26: sk_pq.

The schematic CTP diagram indicates some transaction schema input states for process p_i:

- $\{s^1_{p_i}, s^2_{p_i}, \ldots, s^{m_i}_{p_i}\}$,

by an arrow from the p_i control states to TS_j and some transaction schema output states for process p_i by an arrow from TS_j (back) to the p_i control states. These two sets are usually the same.

- The set of all allowable, i.e., specified state to next state transitions can be specified as a set of triples, each triple being of the form:
  - ⋆ (s, ts_n, s') for process p_i: $(s_{p_i}, ts_n, s'_{p_i})$

where ts_n names a transaction schema and where s and s' belong to a process.

- If ts_n supports processes p_i, p_j, ..., p_k, then there will be triples:
  - ⋆ $(s_{p_i}, ts_n, s'_{p_i})$, $(s_{p_j}, ts_n, s'_{p_j})$, ..., $(s_{p_k}, ts_n, s'_{p_k})$

Figure 13.27 hints at such transition triples.

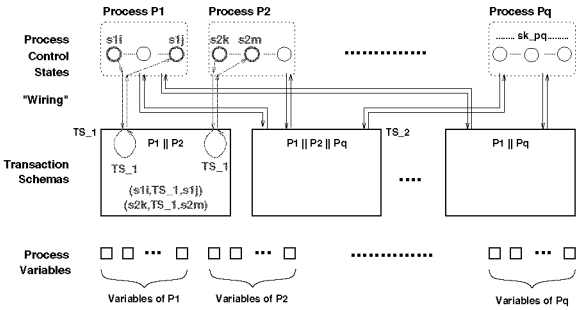

Fig. 13.27. State to next state transitions shown for TS_1 only

CTP Transaction Schemas

Figure 13.28 indicates that a transaction schema consists of one or more transaction charts.

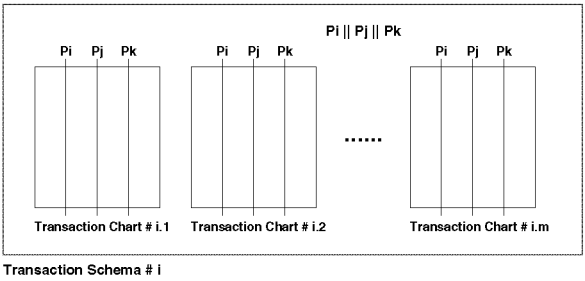

Fig. 13.28. Transaction charts of a transaction schema

Each transaction schema, TS_i, thus contains one or more transaction charts: Ch_i^j (for suitable i's and j's).[2] Each transaction chart contains one simple message sequence chart. Instances (i.e., vertical lines) of the message sequence charts are labelled by distinct process names. All transaction charts of a transaction schema contain message sequence charts whose instances are labelled by the same set of process names.

CTP Transaction Charts

To each transaction chart there is associated a process name indexed set, G_n^j, of propositions for TS_n transaction chart Ch_n^j. See Fig. 13.29.

[2] In Fig. 13.28 Ch_i^j is represented by Transaction Chart # i.j.

Simple CTP Message Sequence Charts

Each instance of each simple message sequence chart of each transaction chart of each transaction schema may contain zero, one or more internal actions,

- $a_i^{j_k}$,

and input/output events:

- $(p_i \leftarrow p_j)?v_i^{\nu}$

(input value offered on channel from process p_j to process p_i and assigned to process p_i's variable v_i^{ν}), respectively,

- $(p_i \rightarrow p_j)!e_i^k$

(output value of expression e_i^k over variables of process p_i from process p_i to process p_j). The variables of respective processes are shown as square boxes in Fig. 13.26.

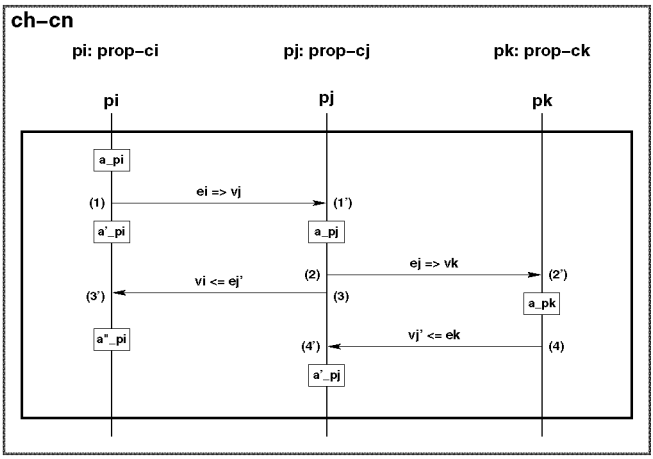

Fig. 13.29. A transaction chart with a simple message sequence chart

Figure 13.29 shows a transaction chart with a simple message sequence chart which prescribes the interaction among three processes, p_i, p_j and p_k. Instance p_i shows the following sequence of events:

- $\langle a_{pi} \ , \ (p_i \rightarrow p_j)!e_i \ , \ a'_{pi} \ , \ (p_i \leftarrow p_j)?v_i \ , \ a''_{pi} \rangle$

The output/input messages (ℓ, ℓ') [or (ℓ', ℓ)], shown as labelled arrows: $\xrightarrow{e \Rightarrow v}$ or $\xleftarrow{v \Leftarrow e}$ correspond to the pairs of (output,input) events in respective processes.

$(1 \xrightarrow{ei \Rightarrow vj} 1')$: the pair of $((p_i \rightarrow p_j)!e_i, (p_j \leftarrow p_i)?v_j)$,

$(2 \overset{ej \Rightarrow vk}{\longrightarrow} 2')$: the pair of $((p_j \rightarrow p_k)!e_j, (p_k \leftarrow p_j)?v_k)$,
$(3' \overset{vi \Leftarrow ej'}{\longleftarrow} 3)$: the pair of $((p_j \rightarrow p_i)!e'_j, (p_i \leftarrow p_j)?v_i)$,
$(4' \overset{vj' \Leftarrow ek}{\longleftarrow} 4)$: the pair of $((p_k \rightarrow p_j)!e_k, (p_j \leftarrow p_k)?v'_j)$.

Enabled CTP Transaction Charts

If the transaction schema is labelled with process names $\{p_i, p_j, p_k\}$ then one transition from control states of each of processes p_i, p_j and p_k leads into each transaction chart of that transaction schema. In order for a transaction chart (of a transaction schema) to be enabled the following two conditions must be fulfilled:

- One each of the input control states of processes p_i, p_j and p_k must be marked. That is, one each of $s^1_{p_i}, s^2_{p_i}, \ldots, s^{m_1}_{p_i}$, and $s^1_{p_j}, s^2_{p_j}, \ldots, s^{m_1}_{p_j}$, and $s^1_{p_k}, s^2_{p_k}, \ldots, s^{m_1}_{p_k}$, must be marked. More precisely, all the control state preconditions of the transaction schema to which this chart belongs are fulfilled.
- The indexed set of propositions for the transaction chart must all evaluate to true.

In the example of the transaction chart of Fig. 13.29 the indexed set of propositions are the three propositions *prop-ci, prop-cj* and *prop-ck*. Each proposition for any process p_i of any transaction chart may contain variables, if so they must only be variables of that process.

Enabled Versus Invoked Schemas and Charts

A distinction is being made between being enabled and being invoked. An invoked schema or chart must be enabled. Enablement means that the conditions for invocation are satisfied. Invocation means that an actual interpretation (i.e., execution) takes place with all attendant state changes possibly occurring.

Details of Invocation and Execution

We elaborate a bit further on the interpretation of a CTP program (i.e., diagram). Initially control rests in the process initial control states. No transaction schema is invoked.

Now zero, one or more transaction schemas may be enabled. For a transaction schema to be enabled the following must hold. One or more of the transaction charts of this transaction schema must be enabled. That is, their guards must hold. That is evaluate to true in the initial state of the process variables. One or more enabled transaction schema may now be invoked provided that no two of them share processes. Invoking an enabled transaction

schema means the following: One of its enabled transaction charts will be nondeterministically selected. To thus invoke an enabled and selected transaction chart means that the marking (i.e., the tokens) of the enabling process control states will be removed and "converted" into an instance ("program point") pointer for each of the process instances of the enabled and selected transaction chart, and those pointers are initially set to zero (0), i.e., the beginning, the "entry", of the transaction chart instances.

The preceding paragraph outlines a step (in this case a zeroth step) of CTP program interpretation (i.e., execution).

Now an interpretation of the instances of the enabled and selected transaction chart takes place. Here we refer to the description of the semantics of BMSCs (basic MSCs) earlier in this chapter. A step is made up from either interpreting an internal action (which usually will update process control variables and hence atomic propositions), or interpreting an output event, or interpreting an input event. The instance program pointers are advanced one position for each such interpretation. When all instance program pointers of a specific transaction chart (of a specific transaction schema) reach their respective last positions, then the transaction chart and its transaction schema are disabled and the designated output control states are marked.

At the same time as a step related to one particular enabled and invoked transaction schema and a transaction chart within it is being performed similar steps may be performed, concurrently, at or within other enabled and invoked transaction schemas and transaction charts within them. So, as an illustration, as one step of interpretation occurs properly within a transaction chart of one transaction schema, another such step of interpretation may occur properly within a transaction chart of another transaction schema, and yet a third transaction chart may be enabled, selected, invoked, and so on.

CTP Transitions

The semantics of CTP calls for transitions from input control states via enabled transaction schemas to (output) control states. Figure 13.27 hinted at such transitions.

An invoked transaction chart will then result in the appropriate input states no longer being marked, in the execution of the simple message sequence chart, from top to bottom, in the updating of process variables (as the result of execution of each of the instances of the simple message sequence chart), and, once message sequence chart execution terminates, in the marking of one appropriate output state for each of the processes labelling that transaction chart.

Which of the output states, for processes p_i, p_j and p_k, that is,

- which of $s'^1_{p_i}, s'^2_{p_i}, \ldots, s'^{m_i}_{p_i}$, and
- which of $s'^1_{p_j}, s'^2_{p_j}, \ldots, s'^{m_j}_{p_j}$, and
- which of $s'^1_{p_k}, s'^2_{p_k}, \ldots, s'^{m_k}_{p_k}$

are selected is determined by which of the

- $(s^{\alpha}_{p_i}, ts_n, s^{\beta}_{p_i})$

transition rules had their

- $s^{\alpha}_{p_i}$

part apply in the invocation of transaction schema ts_n to which this chart belongs.

For technical reasons no two otherwise distinct transition rules $(s^{\gamma}_{p_i}, ts_n, s^{\delta}_{p_i})$ and $(s^{\phi}_{p_i}, ts_n, s^{\psi}_{p_i})$ can have identical first pairs, i.e., $\gamma \neq \phi$, and cannot have identical last pairs, i.e. $\delta \neq \psi$. Thus we assume that each transaction schema ts_n, has exactly one input and one output control state for each process.

The process control states are like places (conditions), and the transaction schemas are like transitions in a condition event Petri net.

Firing (i.e., invocation) means that one or more enabled transaction schemas (that do not share processes) are selected, that is, one or more transaction schemas for which the guards of one or more transaction charts evaluate to true (i.e., is enabled) — and that within each such selected transaction schema one such (enabled) transaction chart is selected (invoked). The invoked transaction charts are then "executed", as would a normal message sequence chart. Once any such message sequence chart execution has completed, the transition completes by marking the designated output control states. Since several transaction schemas may be enabled in this way one or more are chosen nondeterministically. And since within each transaction schema several transaction charts may be enabled one is chosen nondeterministically.

13.6.3 A Dining Philosophers Example

Before we formalise the diagrammatic language of CTPs we bring in an example.

Example 13.24 *Dining Philosophers:* This whole section is one example, but we omit shading. ▪

We model the classical dining philosophers problem using CTP. For simplicity, we consider the setting of just two philosophers. As illustrated in Fig. 13.30, two philosophers $P1$ and $P2$ are seated on opposite sides of a round table and two forks $F1$ and $F2$ are placed between $P1$ and $P2$.
A plate of spaghetti is placed at the centre of the dining table. A philosopher alternates between eating and thinking. To eat the spaghetti, a philosopher must try to grab (the) two forks (here $F1$ and $F2$). And when a philosopher finishes eating, he puts down both forks. The problem is to devise a strategy of using the forks such that the philosophers do not suffer starvation.

The CTP program for the dining philosophers problem is shown in Fig. 13.31.

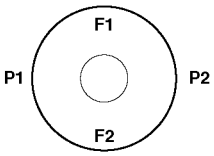

Fig. 13.30. Two dining philosophers table with forks

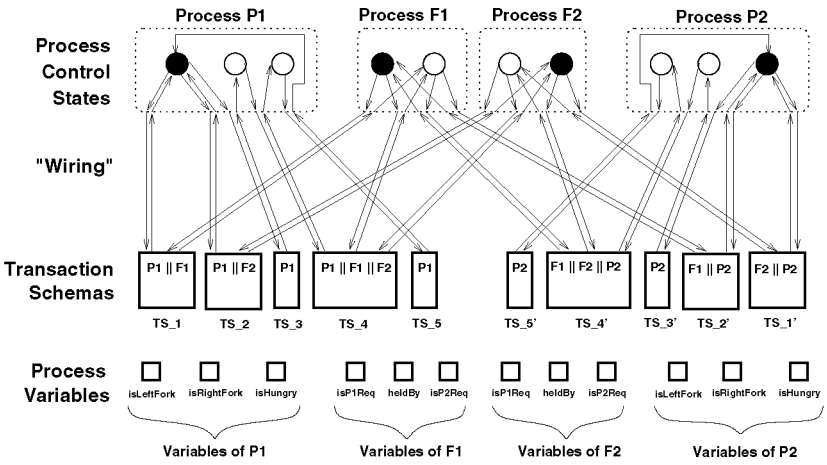

Fig. 13.31. Two dining philosophers CTP program

There are four processes $P1$, $P2$, $F1$ and $F2$ corresponding to the two philosophers and the two forks. In transaction schema TS_1, $P1$ tries to grab its left fork $F1$. In TS_2, $P1$ tries to grab its right fork $F2$. TS_3 represents the behaviour where $P1$ is eating (after getting hold of both forks $F1$ and $F2$). TS_4 represents the behaviour where $P1$ puts down both forks (after finishing eating). Finally, TS_5 models the behaviour where $P1$ is thinking. Analogously, transaction schemas TS_1', TS_2', TS_3', TS_4', TS_5' represent the behaviours where $P2$ tries to grab its left fork $F2$, $P2$ tries to grab its right fork $F1$, $P2$ is eating, $P2$ puts down both forks, and $P2$ is thinking.

The initial control states of each process are shown by darkened places.

The process $P1$ has three variables, isLeftFork, isRightFork and isHungry, all of which are of type **Bool**. These three variables indicate whether $P1$ holds its left fork, whether $P1$ holds its right fork, respectively whether $P1$ is hungry. Initially, $P1$ holds neither fork and is hungry. The variables of $P2$ are set up similarly to $P1$.

The process $F1$ has three variables isP1Req, heldBy and isP2Req. The variable isP1Req (respectively isP2Req) is of type **Bool** and records whether there is a request from $P1$ (respectively $P2$) to hold $F1$. The variable heldBy is an enumerated type variable that takes one of the three values mkNil (meaning

that $F1$ is held by neither philosopher), mkP1 (meaning that $F1$ is held by $P1$) and mkP2 (meaning that $F1$ is held by $P2$). The variables of $F2$ are set up similarly to $F1$.

In Fig. 13.32 we show the transaction charts of TS_1.

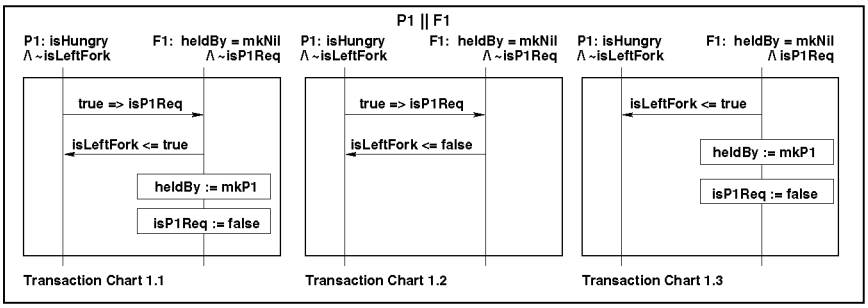

Fig. 13.32. Transaction schema TS_1

There are three transaction charts 1.1, 1.2, 1.3. Chart 1.1 models the scenario that $F1$ grants a fresh "grab" request by $P1$, while chart 1.2 models that $F1$ rejects a fresh "grab" request by $P1$ (but remembers this request). The chart 1.3 models that $F1$ grants a previously recorded request from $P1$. Obviously, the transaction charts of TS_2 are similar to those of TS_1 and thus we omit the details of TS_2.

Transaction schema TS_3 is shown in Fig. 13.33.

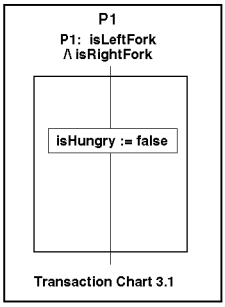

Fig. 13.33. Transaction schema TS_3

Since it only involves $P1$, we would have only internal actions of $P1$. In particular, the activity of eating is modelled by setting isHungry to false.

The transaction schema TS_4 is shown in Fig. 13.34.

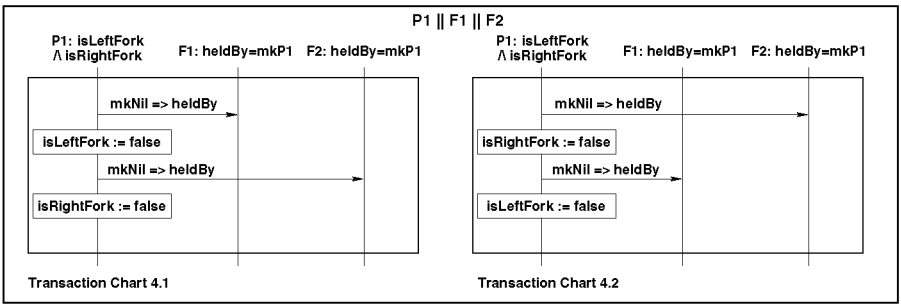

Fig. 13.34. Transaction schema TS_4

There are two charts corresponding to whether $P1$ first puts down its left fork or its right fork.

Similarly to TS_3, the transaction schema TS_5 (shown in Fig. 13.35) models the activity of thinking by setting isHungry to true! Process $F1$ (and also $F2$) alternates between communicating with $P1$ and $P2$. Initially $F1$ is ready to communicate with $P1$ and $F2$ is ready to communicate with $P2$.

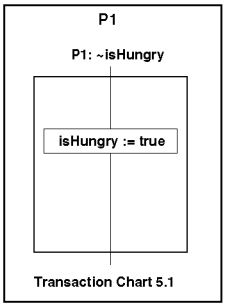

Fig. 13.35. Transaction schema TS_5

We omit the details of TS_1', TS_2', TS_3', TS_4', TS_5' as they are analogous to TS_1, TS_2, TS_3, TS_4, TS_5.

End of Example 13.24. ▪

13.6.4 Formalisation of CTPs

The Syntactic and Some Semantic Types

type

P, T, S, Var, Typ, VAL, Chtn, Exp, AP, Act

Annotation:

P, T, S, Var, Typ, VAL, Chtn, Exp, AP, Act: Process names, transaction schema names, process control states (i.e., names), variable identifiers, type designators (for example `integer, Boolean` and so on), semantic values (for example **Int, Bool** and so on), chart names, expressions (further undefined, but are usually variables, prefix expressions and infix expressions over usual integer operators and Boolean connectives), atomic propositions (i.e., Boolean valued expressions over variables) and internal actions (assignments, conditional actions, etc.). ▪[3]

type
Prog′ = PDecls × TDecls × Wiring × Init
Prog = {| prog:Prog′• wf_Prog(prog) |}

Annotation:

Prog: A CTP program consists of well-formed combinations of process variable and transaction schema declarations, of wiring and the definition of an initialisation (of process control states and variable values). ▪

type
PDecls = P $\overrightarrow{m}$ VarDecl
TDecls = T $\overrightarrow{m}$ (Chtn $\overrightarrow{m}$ (Gd × Cht))

Annotation:

PDecls, VarDecl: For each process there is a set of variables of specified type.
TDecls: For each transaction schema name, T, there is a set of uniquely named, Chtn, transaction charts, with each chart consisting of a guard, Gd, and the chart proper Cht. ▪

type
Wiring = T $\overrightarrow{m}$ (P $\overrightarrow{m}$ S × S)
Init = P $\overrightarrow{m}$ (S × VarInit)
VarDecl = Var $\overrightarrow{m}$ Typ

Annotation:

Wiring: For each transaction schema and for each process (that applies to this schema) there is a pair of respectively input and output control states.
Init, VarInit: With each process a control state, S, is associated an initialisation, respectively the current values of all variables of this process. ▪

type
Gd = P $\overrightarrow{m}$ Prop
Prop == mkTrue | mkAP(ap:AP) | mkNot(pr:Prop)
| mkAnd(pr:Prop,pr′:Prop) | mkOr(pr:Prop,pr′:Prop)

[3] ▪ means: end of annotation.

Annotation:

Gd, Prop: A transaction chart guard associates
- to each of the processes associated with that chart
- a proposition which is
- either the value true,
- or is an atomic proposition,
- or a negated,
- or a conjunctive
- or a disjunctive

proposition. ▪

type
Cht = (P $\overrightarrow{m}$ Ev*) × SendRecv
Ev == mkSe(p:P,e:Exp) | mkRe(p:P,v:Var) | mkAct(act:Act)
SendRecv = (P × Pos) $\overrightarrow{m}$ (P × Pos)
Pos = **Nat**
Σ = Var $\overrightarrow{m}$ VAL
VarInit = Σ

Annotation:

Cht, Ev*, SendRecv: A transaction chart maps each of its associated processes into an instance — which is an event list — and a mapping, SendRecv, that relates output and input events in respective process instances.
Ev: An event is either a send event (sending to p the value of expression e), or a receive event (receiving a value from p and storing it in v); or an event is an internal action.
Pos: A position is an index into an event list. ▪

Auxiliary Syntactic and Semantic Function Signatures

value
typeof: Exp → VarDecl → Typ

wf_AP: AP → VarDecl → **Bool**
wf_Exp: Exp → VarDecl → **Bool**
wf_Act: Act → VarDecl → **Bool**

Annotation:

typeof: Extracts from an expression, given a set of variable declarations, the type of the value of the expression, if well-formed.
wf_AP: Examines whether an atomic proposition is well-formed.
wf_Exp: Examines whether an expression is well-formed.
wf_Act: Examines whether an internal action text is well-formed. ▪

```
value
  eval_AP: AP → Σ → Bool
  eval_Exp: Exp → Σ → VAL
  int_Act: Act → Σ → Σ
```

Annotation:

eval_AP: Evaluates an atomic proposition.
eval_Exp: Evaluates an expression.
int_Act: Interprets an internal action, possibly leading to changes in the values of variables. ∎

Auxiliary Function Signatures and Definitions

```
value
  participants: T → Prog′ → P-set
  participants(t)(prog) ≡ let (_,_,wiring,_)= prog in dom wiring(t) end

  instances : Cht → P-set
  instances(cht) ≡ let (pevs,_) = cht in dom pevs end
```

Annotation:

participants: Extracts the set of process (names) participating in a transaction schema
instances: Extracts the set of instances of a chart. ∎

```
value
  xtr_APs: Prop → AP-set
  xtr_APs(pr) ≡ case pr of mkTrue → {}, mkAP(ap) → {ap}, ... end

  eval_Prop: Prop → Σ → Bool
  eval_Prop(pr)(σ) ≡
    case pr of mkTrue → true, mkAP(ap) → eval_AP(ap)(σ), ... end
```

Annotation:

xtr_APs: Extracts, from a proposition, the set of atomic propositions occurring in a proposition.
eval_Prop: Evaluates a proposition. ∎

Well-formedness of CTP

```
value
  wf_Prog : Prog′ → Bool
  wf_Prog(prog) ≡
    All_Wired(prog) ∧ All_Initialized(prog) ∧
    wf_Gds_and_Chts(prog) ∧ wf_Wiring(prog) ∧ wf_Init(prog)
```

Annotation:

wf_Prog: Conjunction of five constraints. ▪

```
value
  All_Wired: Prog′ → Bool
  All_Wired(_,tdecls,wiring,_) ≡ dom tdecls = dom wiring

  All_Initialized: Prog′ → Bool
  All_Initialized(pdecls,_,_,init) ≡ dom pdecls = dom init
```

Annotation:

All_Wired: All transaction schemas are wired.
All_Initialized: Each process is initialized. (The initialization of a process includes not only the variables but also an initial control state.) ▪

```
value
  wf_Gds_and_Chts: Prog′ → Bool
  wf_Gds_and_Chts(prog) ≡
    let (pdecls,tdecls,_,_) = prog in
    ∀ t:T•t ∈ dom tdecls ⇒
      let (gd,cht) = tdecls(t)(chtn) in
      dom gd = instances(cht) = participants(t)(prog) ∧
      wf_Gd(gd)(pdecls) ∧ wf_Cht(cht)(pdecls)
    end end

  wf_Gd: Gd → PDecls → Bool
  wf_Gd(gd)(pdecls) ≡
    ∀ p:P•p ∈ dom gd ⇒ ∀ ap:AP • ap ∈ xtr_APs(gd(p))
      ⇒ wf_AP(ap)(pdecls(p))
```

Annotation:

wf_Gds_and_Chts: The guards and charts are well-formed.
wf_Gd: Examines whether a guard is well-formed. ▪

```
value
  wf_Cht: Cht → PDecls → Bool
  /∗ see later ∗/

  wf_Wiring: Prog′ → Bool
  wf_Wiring(prog) ≡
    let (pdecls,_,wiring,_) = prog in
      ∀ t:T•t ∈ dom wiring ⇒
      participants(t)(prog)⊆dom pdecls
    end
```

Annotation:

wf_Wiring: The wiring is well-formed. ▪

```
value
wf_Init: Prog′ → Bool
wf_Init(prog) ≡
  let (pdecls,_,_,init) = prog in
  ∀ p:P•p ∈ dom init ⇒
    let (s,varinit) = init(p) in
    (∃ t:T,s′:S • (s,s′)=wiring(t)(p)) ∧ wf_VarInit(varinit)(vardecl(p))
  end end
```

Annotation:

wf_Init: The initialisation is well-formed (the initialisation includes both initial control states and initial values of variables). ▪

```
value
  wf_VarInit: VarInit → VarDecl → Bool
  wf_VarInit(varinit)(vardecl) ≡
    dom vardecl = dom varinit ∧
    ∀ var:Var•var ∈ dom vardecl ⇒
      typeof_VAL(varinit(var))=vardecl(var)

  typeof_VAL: VAL → Typ
```

Annotation:

wf_VarInit: All variables are initialised to values of the declared type.
typeof_VAL: Similar to typeof. ▪

Well-formedness of Charts

```
value
  wf_Cht: Cht → PDecls → Bool
  wf_Cht(cht)(pdecls) ≡ wf_Evs(cht)(pdecls) ∧ wf_SendRecv(cht)(pdecls)
```

Annotation:

wf_Cht: All events are well-formed and so are all send-receive pairs. ▪

```
value
  wf_Evs: Cht → PDecls → Bool
  wf_Evs(pevs,_)(pdecls) ≡
    ∀ p:P,ev:Ev•
      p ∈ dom pevs ∧ ev ∈ elems pevs(p) ⇒
        case ev of
```

```
    mkSe(q,exp)→q ∈ dom pevs\{p}∧wf_Exp(exp)(pdecls(p)),
    mkRe(q,var)→q ∈ dom pevs\{p}∧is_decl(var)(pdecls(p)),
    mkAct(act)→wf_Act(act)(pdecls(p))
end
```

Annotation:

wf_Evs: All events are well-formed (with respect to source/target processes, expressions, etc.)

- Sends and receives are between different instances, that is, processes.
- Corresponding expressions are well-formed and corresponding variables are declared.
- Internal actions are well-formed. ∎

```
value
  is_decl: Var → VarDecl → Bool
  is_decl(var)(vardecl) ≡ var ∈ dom vardecl

  wf_SendRecv: Cht → PDecls → Bool
  wf_SendRecv(cht)(pdecls) ≡
    Well_Matched(cht)(pdecls) ∧ All_Matched(cht) ∧ ∼is_cyclic(cht)
```

Annotation:

is_decl: Examines whether the variable is properly declared.
wf_SendRecv: The send-receive matching relation is well-formed. ∎

```
value
  is_cyclic: Cht → Bool
  is_cyclic(cht) ≡ ... /∗ straightforward ∗/
```

Annotation:

is_cyclic: The transitive closure of the send-receive and instancewise message ordering relation contains cycles. (The specification of this predicate is clear from item /*6*/ (Page 387) of Sect. 13.1.6 "Syntactic Well-formedness of MSCs".) ∎

```
value
  Well_Matched: Cht → PDecls → Bool
  Well_Matched(pevs,sendrecv)(pdecls) ≡
    card dom sendrecv = card rng sendrecv ∧
    ∀ (p,i),(q,j):P×Pos • sendrecv((p,i))=(q,j) ⇒
      ∃ exp:Exp,var:Var•
        pevs(p)(i) = (q,exp) ∧
        pevs(q)(j) = (p,var) ∧
        typeof(exp)(pdecls(p)) = pdecls(q)(var)
```

Annotation:

Well_Matched: The matching is proper. ▪

```
value
  All_Matched: Cht → Bool
  All_Matched(pevs,sendrecv) ≡
    dom sendrecv = {(p,i)|(p,i):P×Pos • is_Send_Ev(pevs(p)(i))}
```

Annotation:

All_Matched: All send/receive events are matched. ▪

```
value
  is_Send_Ev: Ev → Bool
  is_Send_Ev(ev) ≡ case ev of mkSe(_,_) → true, _ → false end
```

Annotation:

is_Send_Ev: Examines whether an event is a send event. ▪

Dynamic Semantics, Types

Semantic Types

```
type
  PΨ = P ~m~> Ψ
  Ψ = Π × Σ × Θ
```

Annotation:

PΨ : The current "stage" of a CTP program is given by associating with each process, a stage, Ψ.

Ψ: The process stage consists of a triple: the current program point, Π, the current values of all its variables, Σ, and the (evaluated) values of expressions of executed output (send) events, Θ. ▪

```
type
  Π == mkS(s:S) | mkT(t:T,chtn:Chtn,i:Pos)
  Θ = Pos ~m~> VAL
  Pos = Nat
```

Annotation:

Π : The program pointer (of a process) either designates a process control state mkS(s:S) or a position i:Pos within a transaction chart chtn:Chtn of a transaction schema t:T; i=0 indicates that the process has just entered the chart.

Θ: The output value queue (of executed output events) is a map from positions, Pos, of output events to values VAL.
Pos: Position of events (input/output events and internal actions). ▪

type
$\mathsf{P}\Delta = \mathsf{P} \xrightarrow{m} \Delta$

Annotation:

$\mathsf{P}\Delta$: For each (invoked) process P we record its stepwise progress Δ. ▪

type
$\Delta = \mathrm{T} \times \mathrm{Chtn} \times \Phi$
Φ == mkEnter | mkEv(i:Pos) | mkExit

Annotation:

Δ : The stepwise progress within a transaction chart, Chtn, of a transaction schema, T, is recorded by a quantity Φ.
Φ : Either the process, at an instance, is at the point of entering, mkEnter, or leaving, mkExit, or is at some event position, mkEv(i:Pos). ▪

Well-formedness

value
wf_PΔ: PΔ → Prog → **Bool**
wf_PΔ(pδ)(prog) ≡
let (pdecls,_,_,_) = prog **in**
dom pδ ⊆ **dom** pdecls ∧
∀ p:P•p ∈ **dom** δ ⇒ wf_Δ(p)(pδ)(prog)
end

Annotation:

wf_PΔ :
- The invoked processes must first have been declared.
- And for each such process its progress must be well-formed. ▪

value
wf_Δ: P → PΔ → Prog → **Bool**
wf_Δ(p)(pδ)(prog) ≡
let (pdecls,tdecls,_,_) = prog, (t,chtn,ϕ) = pδ(p) **in**
t ∈ **dom** tdecls ∧ chtn ∈ **dom** tdecls(t) ∧ p ∈ participants(t)(prog) ∧
case ϕ **of**
mkEv(i)
→ **let** (pevs,_) = tdecls(t)(chtn) **in** i ∈ **inds** pevs(p) **end**
_ → ∀ q:P•q ∈ participants(t)(prog) ⇒ pδ(q) = pδ(p)
end end

Annotation:

wf_Δ : For the invoked process
- the designated transaction schema and transaction chart (of that schema) must be declared, and the designated process (name) must be an instance of that chart.
- In addition the program point (ppt) must be well-formed:
  - ⋆ if an event index it must be into the process instance, otherwise
  - ⋆ all processes of that transaction chart must be in the same (either entry or exit) state. ▪

Dynamic Semantics, Functions

Auxiliary Functions

```
value
  xtr_preS: Prog → T → P → S
  xtr_preS(_,_,wiring,_)(t)(p) ≡
    let (s,_) = wiring(t)(p) in s end
    pre t ∈ dom wiring ∧ p ∈ dom wiring(t)
```

Annotation:

xtr_preS : Extract from a transaction schema, the precondition (a control state) corresponding to a process. ▪

```
value
  xtr_postS: Prog → T → P → S
  xtr_postS(_,_,wiring,_)(t)(p) ≡
    let (_,s) = wiring(t)(p) in s end
    pre t ∈ dom wiring ∧ p ∈ dom wiring(t)
```

Annotation:

xtr_postS : Given a
- program, a transaction schema (name) and a process (name)
- yield the output control state (from the wiring). ▪

```
value
  xtr_Ev: Prog → (T × Chtn × P × Pos) → Ev
  xtr_Ev(_,tdecls,_,_)(t,chtn,p,i) ≡
    let (_,(pevs,_)) = tdecls(t)(chtn) in pevs(p)(i) end
    pre t ∈ dom tdecls ∧ chtn ∈ dom tdecls(t) ∧
        let (_,(pevs,_)) = tdecls(t)(chtn) in
        p ∈ dom pevs ∧ i ∈ inds pevs(p) end
```

Annotation:

xtr_Ev : Given
- a program,
- a transaction schema name (within that program),
- the name of a chart (within that schema),
- a process (name) and
- a position (within the designated chart),

yield the designated event. ▪

value
```
xtr_Prop: Prog → (T × Chtn) → P → Prop
xtr_Prop(_,tdecls,_,_)(t,chtn)(p) ≡
  let (gd,_) = tdecls(t)(chtn) in gd(p) end
  pre t ∈ dom tdecls ∧ chtn ∈ dom tdecls(t) ∧
      let (_,cht) = tdecls(t)(chtn) in p ∈ instances(cht) end
```

Annotation:

xtr_Prop :
- Given
  - ⋆ a program,
  - ⋆ a transaction schema name (within that program),
  - ⋆ the name of a chart (within that schema), and
  - ⋆ a process (name)
- yield the designated proposition. ▪

value
```
last_Pos: Prog → (T × Chtn) → P → Pos
last_Pos(_,tdecls,_,_)(t,chtn)(p) ≡
  let (_,(pevs,_)) = tdecls(t)(chtn) in len pevs(p) end
  pre t ∈ dom tdecls ∧ chtn ∈ dom tdecls(t)∧
      let (_,cht) = tdecls(t)(chtn) in p ∈ instances(cht) end
```

Annotation:

last_Pos :
- Given
  - ⋆ a program,
  - ⋆ a transaction schema (name, within that program),
  - ⋆ a chart (name, within that schema), and
  - ⋆ a process (name, within that chart)
- yield the position of the last event of the designated process instance. ▪

value
```
xtr_Send: Prog → (T × Chtn) → (P × Pos) → (P × Pos)
xtr_Send(_,tdecls,_,_)(t,chtn)(p,i) as (q,j)
  pre  t ∈ dom tdecls ∧ chtn ∈ dom tdecls(t) ∧
       let (_,(pevs,_))=tdecls(t)(chtn) in
```

```
        p ∈ dom pevs ∧ i ∈ inds pevs(p) end
    post let (_,(_,sendrecv))=tdecls(t)(chtn) in
        sendrecv((q,j)) = (p,i) end
```

Annotation:

xtr_Send : Extract the matching send event, given a receiving event.

- The transaction schema and chart names must be declared and the event position be appropriate.
- The matching send event (q,j) is then found from the send-receive mapping. ▪

Initialization

```
value
  init_PΨ: Prog → PΨ
  init_PΨ(prog) ≡
    let (_,_,_,init) = prog in
    [p↦convert_Ψ(init(p))|p:P•p ∈ dom init] end

  convert_Ψ: (S × VarInit) → Ψ
  convert_Ψ(s,varinit) ≡ (mkS(s),varinit,[])
```

Annotation:

init_PΨ : To initialise a program is to create the collection of all process initial states.

convert_Ψ : Mark the initial control state, use the initial control variable values and set the initial queues of values of expression of send events to empty. ▪

Enabling

```
value
  is_enabled: PΔ → (Prog × PΨ) → Bool
  is_enabled(pδ)(prog,pψ) ≡
    ∀ p:P•p ∈ dom pδ ⇒ let (t,chtn,φ) = pδ(p) in
      case φ of
        mkEnter → is_enabled_Enter_Chtn(t,chtn)(prog,pψ),
        mkExit → is_enabled_Exit_Chtn(t,chtn)(prog,pψ),
        mkEv(i) → is_enabled_Ev(t,chtn,p,i)(prog,pψ)
      end end
    pre wf_PΔ(pδ)(prog)
```

Annotation:

is_enabled : A program step, pδ, is enabled at the current stage of the program, if every process step corresponding to processes in the domain of this program step is enabled. ▪

```
value
  is_enabled_Enter_Chtn: (T × Chtn) → (Prog × PΨ) → Bool
  is_enabled_Enter_Chtn(t,chtn)(prog,pψ) ≡
    ∀ p:P•p ∈ participants(t)(prog) ⇒
      let s = xtr_preS(prog)(t)(p),
          pr = xtr_Prop(prog)(t,chtn)(p),
          (π,σ,_) = pψ(p) in
      (π=mkS(s)) ∧ eval_Prop(pr)(σ) end
```

Annotation:

is_enabled_Enter_Chtn : A chart of a transaction schema can be entered if for every process participating in this transaction schema, its current control state is the precondition of this transaction schema, and the proposition associated with this process in the guard associated with this chart evaluates to true with respect to the current values of variables. ▪

```
value
  is_enabled_Exit_Chtn: (T × Chtn) → (Prog × PΨ) → Bool
  is_enabled_Exit_Chtn(t,chtn)(prog,pψ) ≡
    ∀ p:P•p ∈ participants(t)(prog) ⇒
      let (mkT(t,chtn,i),σ,_)=pψ(p) in i=last_Pos(prog)(t,chtn)(p) end
```

Annotation:

is_enabled_Exit_Chtn : A chart of a transaction schema can be exited if for every process participating in this transaction schema, it has executed all its events in this chart. ▪

```
value
  is_enabled_Ev: (T × Chtn × P × Pos) → (Prog × PΨ) → Bool
  is_enabled_Ev(t,chtn,p,i)(prog,pψ) ≡
    let (mkT(t,chtn,i′),_,_) = pψ(p) in i′=i−1 ∧
    case xtr_Ev(prog)(t,chtn,p,i) of
      mkRe(q,_) →
        let (q,j) = xtr_Send(prog)(t,chtn)(p,i) in
        let (mkT(t,chtn,j′),_,_) = pψ(q) in j ≤ j′ end end
      _ → true
    end end
```

Annotation:

is_enabled_Ev : An event at a position of a process in a chart of a transaction schema is enabled, if this process has come to the previous position, and in case this event is a receive event, the matching send event has been executed. ▪

Firing

```
value
  fire: (Prog × PΨ) → PΔ → (Prog × PΨ)
  fire(prog,pψ)(pδ) as (prog,pψ′)
    pre is_enabled(pδ)(prog,pψ)
    post pψ′=pψ†[ p↦upd_Ψ(prog,pψ)(pδ)(p)|p ∈ dom pδ ]
```

Annotation:

fire : Firing an enabled program step updates the current stage of every process. ∎

```
value
  upd_Ψ: (Prog × PΨ) → PΔ → P → Ψ
  upd_Ψ(prog,pψ)(pδ)(p) ≡
    let (π,σ,θ) = pψ(p), (t,chtn,φ) = pδ(p) in
      case φ of
        mkEnter → (mkT(t,chtn,0),σ,[ ])
        mkEv(i) →
          let σ′ = upd_Σ(prog,σ)(p)(t,chtn,i),
              θ′ = upd_Θ(prog,θ)(p)(t,chtn,i) in
          (mkT(t,chtn,i),σ′,θ′) end
        mkExit → let s = xtr_postS(prog)(t)(p) in (mkS(s),σ,[ ]) end
    end end
  pre ...
```

Annotation:

upd_Ψ : Upon firing an enabled program step, the current stage of a process should be updated as follows.

- If this process enters a chart of a transaction schema, then this process goes to position zero of this chart (in this transaction schema), retains the current values of variables and initializes an empty map of positions to values of expressions of send events.
- If this process executes an event at a position of a chart of a transaction schema, then this process goes to this position and updates the current values of variables and the map of positions to values of expressions of send events.
- If this process exits a chart of a transaction schema, then this process goes to the postcondition associated with this process of this transaction schema, retains the current values of variables and empties the map of positions to values of expressions of send events. ∎

```
value
  upd_Σ: (Prog × PΨ) → P → (T × Chtn × Pos) → Σ
  upd_Σ(prog,pψ)(p)(t,chtn,i) ≡
```

```
    let (_,σ,_) = pψ(p), ev = xtr_Ev(prog)(t,chtn,p,i) in
    case ev of
      mkSe(q,exp) → σ,
      mkRe(q,var) →
        let (_,_,θ) = pψ(q),
        (q,j) = xtr_Send(prog)(t,chtn)(p,i) in σ † [ var ↦ θ(j) ] end,
      mkAct(act) → int_Act(act)(σ)
    end end
  pre ...
```

Annotation:

upd_Σ : Upon execution of an event, the current values of variables should be updated as follows.

- Executing a send event does not change the value of any variable.
- Executing a receive event amounts to assigning the value of the expression of the matching send event to the variable associated with this receive event, and leaving the values of all other variables untouched.
- Executing an internal action amounts to evaluating it with respect to the current values of variables, possibly leading to changes in the values of variables. ▪

```
value
  upd_Θ: (Prog × PΨ) → P → (T × Chtn × Pos) → Θ
  upd_Θ(prog,pψ)(p)(t,chtn,i) ≡
    let (_,σ,θ) = pψ(p) in
    case ev of mkSe(q,exp) → θ ∪ [i ↦ eval_Exp(exp)(σ)],
                          _ → θ end end
  pre ...
```

Annotation:

upd_Θ : Upon execution of an event, the map of positions to values of expression of send events is updated as follows. Executing a send event amounts to adding to this map the value of the expression of this send event associated with its position. Executing a receive event or an internal action does not touch this map. ▪

13.7 Discussion

13.7.1 General

We have covered two notions of sequence charts (SCs): Message SCs (MSCs) and Live SCs (LSCs).

MSCs arose in connection with the design of software for telephone switching and data communication systems in the 1970s. MSC, as a language, was then known as the System Description Language (SDL). Work on SDL and MSC took place mainly under the auspices of the International Telecommunication Union, ITU. We refer to various URLs related to ITU, SDL and MSC [226].

MSCs, most likely due to the influence of Ivar Jacobson, one of the three technologists who did the principal design of UML[4] [59, 237, 382, 440], became one of the many diagrammatic facets of UML.

LSCs took off from MSC. On one hand, David Harel and his colleagues (notably Werner Damm), have spearheaded LSC research and development, notably through the *Come, Let's Play — Scenario-Based Programming Using LSCs and the Play-Engine* [195]. On the other hand, it seems that, so far, the mostly software oriented computer science community has been at work on studying LSCs. The author happily confesses: The *Play-Engine* is a fascinating concept.

We have also presented some material on theoretical foundations of MSCs and LSCs. The material presented in Sects. 13.3–13.5 represents one direction of research in the field of integrated formal methods. It is included to illustrate that certain techniques have advantages for certain applications in software engineering, and that choosing one technique (e.g., diagrams) does not preclude also using other techniques (e.g., formal specification in RSL). Indeed, in complex software engineering projects, several techniques will be needed to specify all the relevant aspects of a system. To ensure consistency between the different parts of the system specified using different techniques, relations among these techniques must be established. The relation between LSCs and RSL presented above and the corresponding relation between statecharts and RSL — presented in the next chapter (Sect. 14.7) — are two such examples.

13.7.2 Principles, Techniques and Tools

This chapter has basically covered a tool: The sequence charts (MSCs and LCSs). As such we can hardly speak of 'A Principle of Sequence Charts' — such as we could for most other chapters' title subjects. So we shall rearrange things a bit in this section on "Principles, Techniques and Tools".

Principles. *Choosing Sequence Charts:* Sequence charts, as a modelling device, can be chosen when the phenomenon to be abstracted and modelled exhibits concurrent and interacting behaviours, where the interaction "patterns" are of main interest, and then usually when there is a definite, "small" number of behaviours, typically less than a couple of dozen. ▪

[4] Ivar Jacobson was with Ericsson in the later 1970s and early 1980s when SDL was first designed, and played a decisive role in that effort.

Which kind of sequence chart, whether MSCs or LSCs are chosen, is then a matter of sophistication, whether MSCs will do, or whether the more elaborate properties of LSCs are needed. Please contrast the above principle with that of *Choosing Statecharts* Sect. 14.8.2.

Techniques. *Creating Sequence Charts:* The basic parts of sequence charts are the instances, corresponding to behaviours (i.e., processes), and the inputs/outputs, corresponding to events (in the CSP jargon). All else are adornments. ▪

Tools. *Sequence Charts:* We refer to [226] for reference to MSC tools. The main, and overwhelmingly sophisticated, LSC tool is that of the *Play-Engine* [195]. A number of research investigation and exploratory tools are provided by Sun and Dong [493] for model-checking LSCs via translation to CSP and then using the FDR2 tool [442]. Others are provided by Wang, Roychoudhury, Yap and Choudhary [525] for symbolically executing LSCs using translation to constraint logic programming. There are many more. ▪

13.8 Bibliographical Notes

The basic references to MSCs are the three consecutive recommendations from the International Telecommunication Union, labelled Z.120 [227–229] (1992, 1996 and 1999). Syntax recommendations for MSCs are given in Reniers [423]. Extensions of MSCs with time have been studied in [38, 280, 296].

The basic reference to LSCs is Damm and Harel's paper [89]. The main text on LSCs is now the book [195]. A delightful presentation of MSCs and LSCs is Harel and Thiagarajan [199]. The literature on Live Sequence Charts is emerging. A sample is: [58, 64, 89, 187, 191, 278, 493, 525]. In [191] Harel, Kugler and Pnueli put forward further proposals for time in LSC, i.e., the "rich version" of LSC. Report [493] shows relations between the language of LSCs and CSP, and reports on translations of LSCs into CSP for purposes of using CSP's model checker FDR2 `http://www.fsel.com/fdr2_manual.html` [442]. [525] shows how LSCs can be "symbolically executed" using constraint logic programming. Christian Krog Madsen [316, 317] analyses both MSCs, HMSCs and LSCs, establishes proper semantics for these and relates LSCs to RSL. UML contains various rudiments of MSCs [59, 237, 382, 440].

Recent work by Roychoudhury and Thiagarajan merges ideas of Petri nets and MSCs. The result is called *communicating transaction processes* (CTP) [439] — and was treated in Sect. 13.6.

A flurry of recent publications explore various uses of live sequence charts in software engineering and in biology! They are all authored or coauthored by D. Harel. Some recurrent coauthors are I.R. Cohen, S. Efroni, N. Kam, H. Kugler, R. Marelly and A. Pnueli [106–108, 115, 116, 133, 176–180, 182–184, 186, 188–192, 194, 196, 253–258, 279, 325].

13.9 Exercises

Exercise 13.1 *Automatic Teller Machine.* Automatic teller machines (ATM) usually services credit and cash cards of a consortium of financial institutions (Diners, Mastercard, Visa, etc., as well as Citibank (New York, NY, USA), HSBC (Hong Kong and Shanghai Bank Corporation, London, UK), etc.). So you may think of four sets of "players": You, the card holders, the ATMs, the consortia, and the specific financial institutions of the consortia. A particular ATM is bound to a specific set of card types, one consortium, and a specific set of financial institutions. An ATM usually offers a number of services: cash withdrawal, cash deposit, cash transfer, inquiry about account status, etc. An example protocol for the opening of a card transaction using an ATM is as follows: user inserts card into the ATM; the ATM requests card password from the user; the user keys password into the ATM; the ATM requests verification from the consortium; the consortium requests verification from the financial institution of the card; the financial institution either OKs or does not OK the transaction and so informs the consortium; the consortium passes the verification response back to the ATM; and the ATM passes it back to the user. If response was OK, the user may continue.

Exercise 13.1.1: Develop an appropriate MSC for a quadruplet of *User*, *ATM*, *Consortium* and *Financial Institution* instances.

Exercise 13.1.2: Develop a possible MSC, following a successful, i.e., OK'ed verification opening, for a cash withdrawal transaction.

Exercise 13.1.3: Develop a possible MSC, following a successful, i.e., OK'ed verification opening, for a cash transfer transaction.

You are to fill in all relevant details left out above and to take into account that the user makes mistakes.

Exercise 13.2 *Two-Phase Commit Protocol.* In many forms of distributed systems, the need arises for a group of parties to reach an agreement to perform some action. Each party has the option of vetoing the action, in which case all the other parties must not perform the action. Another possibility is that one or more parties fail before either committing or vetoing the action. In that case, the action must also be aborted by all parties.

One application of this protocol is to implement distributed transactions. In this case, the parties must agree whether to commit or roll back the transaction, such that it is either performed by all parties or by none.

The protocol to be formalised is centralised, since a single distinguished party acts as the coordinator. The remaining parties are slaves (A and B).

The informal description (given below) derives from [147] and is based on [469]. Based on this informal description you are to solve the following problems.

Exercises 13.2.1–5: Formalise interactions between the environment, the coordinator and the slaves in terms of live sequence charts.

Exercise 13.2.6: Formalise the internal behaviour of the coordinator in terms of a finite state machine.
Exercise 13.2.7: Formalise all of the above in RSL.

The informal description of our version of the two-phase commit protocol goes as follows, one description part per live sequence chart:

1. *Protocol initiation:* To start the whole thing it is assumed that an environment requests the coordinator to set up requests to all slaves (here just two). Once that *assumption* (modelled in terms of a prechart) is satisfied, the coordinator in any order sends requests to all (i.e., both) slaves.
2. *Commit: When* all (both) slaves commit to the requests (by sending such messages to the coordinator) the coordinator informs the slaves that the protocol has been successfully opened (by sending appropriate messages to the slaves).
3. *Abort by slave A: If* slave A cannot participate in the protocol (i.e., send an abort message), but slave B can (i.e., commits), then the coordinator has to inform slave B of the abort.
4. *Abort by slave B:* Vice versa: *If* slave B cannot the protocol (i.e., aborts), but slave A can (i.e., commits), then the coordinator has to inform slave A of the abort.
5. *Abort by both slaves: If* all (i.e., both) slaves cannot participate in the protocol (i.e., abort), then the coordinator has to inform all slaves of the abort.

It is suggested that the *assumption*, the *when* and the *if*'s of the above five cases be modelled by precharts.

6. *Internal behaviour of coordinator:*

Coordination evolves around a finite state machine. In each state the coordinator expects input (i.e., messages) from either (initially) the environment, or, subsequently, from the slaves. In response to an input the coordinator sends outputs that amount to messages being sent to some or all slaves.

In an initial state the coordinator will expect the environment, i.e., a user to request some action to be performed as a distributed transaction. Once the coordinator receives such a request it is passed on to, i.e., transmitted to the slaves. The coordinator now waits for responses from the slaves. The coordinator can only receive one response at a time. Either a commit from some slave or an abort from some slave. The coordinator, upon receiving one commit or one abort enters respective states in order to be able to receive, distinguish and properly react to subsequent responses from remaining slaves. If all slaves responds with commit, the transaction is committed. If at least one slave responds with abort, the whole transaction is aborted.

7. *An RSL Model:*

You are to model the protocol in RSL. More specifically, to define a number of processes for the system, for example, the coordinator and the two slaves. The system process is suggested to be just the parallel composition of the coordinator process and the two slave processes.

The coordinator process will wait to be invoked by inputting a request from the user. The requested action is transmitted to the two slaves. Next, the coordinator will input the responses from slave A and B, in some order. If both choose to commit, they are informed that agreement has been reached to commit. A function *commit-action* can be postulated to abstract the actual action to be performed. If either slave responds with abort, the other slave is informed that the transaction is aborted and the coordinator performs the necessary clean-up, abstracted, for example, by a function *abort-action*.

The slave processes are entirely analogous. They first wait for a request to be received from the coordinator. Upon receipt, they decide – internal nondeterminism choice – to commit or abort. In the latter case, they tell the coordinator to abort and perform the necessary clean-up, abstracted by *abort-action*. In the former case, they tell the coordinator to commit and await the response. Based on the coordinator's response, they either commit or abort the transaction. The nondeterministic choice is an abstraction of the process used to decide whether to commit or abort.

Exercise 13.3 *Semantics of HMSCs.* We refer to the syntax and well-formedness of BMSCs (Sects. 13.1.2 and 13.1.6), the semantics of BMSCs (Sect. 13.1.8), the syntax and well-formedness of HMSCs (Sects. 13.1.4 and 13.1.6).

Please define the semantics of HMSCs based on the formalisation given in the above referenced sections.

Exercise 13.4 *Remote Procedure Calls/Remote Method Invocation and Broker Design Pattern.*

Procedure calls are a fundamental notion in most imperative computer languages. A procedure call occurs when the calling procedure requests the execution of the behaviour of the body of the called procedure. Typically, the called procedure returns some value once (and if) its execution terminates. A prerequisite for procedure calls is that the caller and callee are contained in the same executable or in shared libraries, which are linked in at runtime. The extended notion of *remote procedure calls (RPC)* does away with this limitation by allowing the caller and callee procedures to be contained in different executables, processes and even on different computers. In the context of Java, RPC is called *remote method invocation (RMI)*. The basic principle of "remoting" is to replace the callee with a proxy procedure, which exposes the same interface as the callee. Instead of performing the action the callee would, the proxy encodes the parameters it is passed, sends them to another proxy, which decodes the parameters and calls the real callee. Figure 13.36 illustrates the setup, where the caller is named the *Client*, while the callee is named the *Server*. The *ClientProxy* appears to the Client as the Server would

(i.e., it has the same interface). The *ServerProxy* appears to the Server as the Client would.

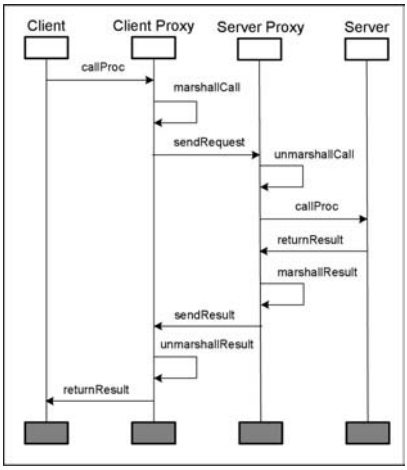

Fig. 13.36. A remote procedure call protocol

The purpose of the two *Proxy* objects is to hide the details of the transmission of the call parameters and return value over some medium, which could be a network connection (for processes distributed on separate computers) or shared memory (for inter-process communication within the same computer). The operation of encoding a call including its parameters is traditionally called *marshalling*. The inverse operation of decoding a call with parameters is traditionally called *unmarshalling*.

Exercise 13.4.1: Formulate an `RSL` specification of a simple `RPC` mechanism for a procedure (function) that takes two integers as arguments and returns their sum. Create a type to represent the marshalled format of the arguments.

A downside with `RPC` is that once the program is written, the Client is locked to the Server. Suppose now that a more efficient implementation of the addition function is written (call it Server′). To make Client call Server′ instead of Server, Client must be modified. Thus this system works best in a static environment, where both Client and Server are developed at the same time. It can not easily adapt to a dynamic environment. One mechanism to introduce dynamic binding of the Client and Server is captured in the so-called *Broker Design Pattern* [71], which is central to `CORBA` [381] and `Java Jini` (`Jini extensible remote invocation (Jini ERI)`) [494] technology. The idea is to introduce a broker, which maintains a list of available services in a distributed system. In the broker architecture, the ServerProxy will register

itself as a service with the broker and provide some form of identification of what the service does. When the ClientProxy is called, it will query the broker to find the location of a service that performs the action it needs. The broker will return some form of address or pointer to the ServerProxy. At this point, the rest of the protocol behaves like the RPC protocol. Hence, once the broker has pointed the ClientProxy to the service, it no longer participates in the communication.

Typically a service is identified by a text string, so the ClientProxy will ask for a service called Addition.

Exercise 13.4.2: Extend the MSC in Fig. 13.36 to include the broker.
Exercise 13.4.3: Extend the previous RSL specification to include the broker.

The interested reader may like to compare the above description to the building blocks of Web services such as XML [417,443,478,546], SOAP [516],WSDL [517] and UDDI [506].

Exercise 13.5 *Generalised Dining Philosophers.* We refer to Sect. 13.6.3. The example of that section showed a two-dining philosophers CTP program. Please show a CTP program solution for five dining philosophers