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.

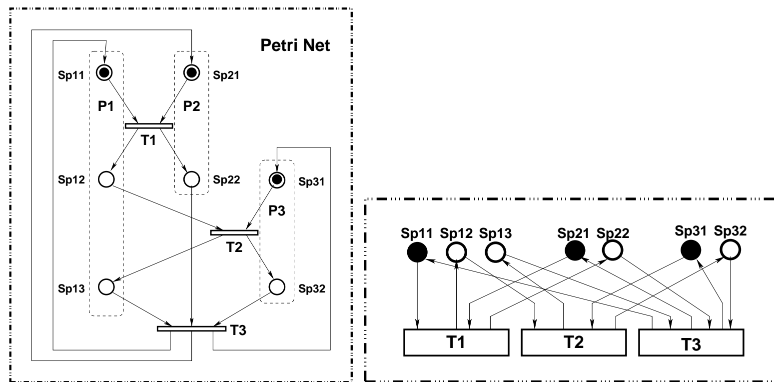

Fig. 13.25. Left: a Petri net. Right: a concrete CTP diagram

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.)

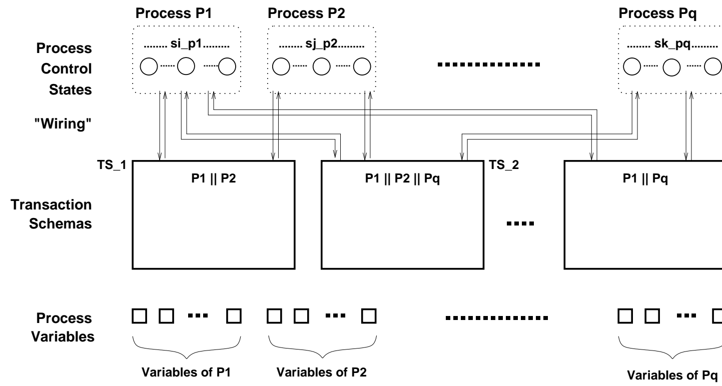

Fig. 13.26. A schematic CTP diagram

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.

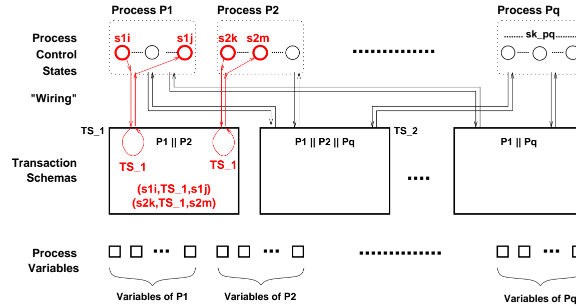

Fig. 13.27. State to next state transitions shown for TS_1 only

CTP Transaction Schemas

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

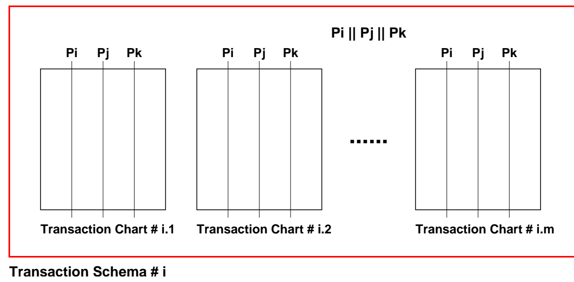

Fig. 13.28. Transaction charts of a transaction schema

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.

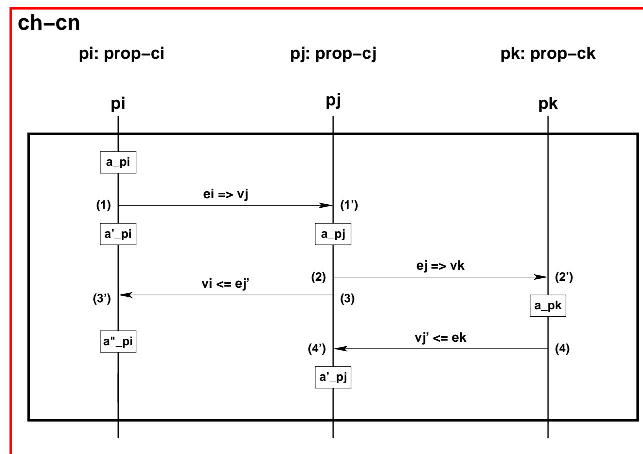

Fig. 13.29. A transaction chart with a simple message sequence chart

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: $\stackrel{e \Rightarrow v}{\longrightarrow}$ or $\stackrel{v \Leftarrow e}{\longleftarrow}$ correspond to the pairs of (output,input) events in respective processes.

$(1 \stackrel{ei \Rightarrow vj}{\longrightarrow} 1')$: 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 $((p_j \to p_k)!e_j, (p_k \leftarrow p_j)?v_k)$,
$(3' \overset{vi \Leftarrow ej'}{\longleftarrow} 3)$: the pair of $((p_j \to p_i)!e'_j, (p_i \leftarrow p_j)?v_i)$,
$(4' \overset{vj' \Leftarrow ek}{\longleftarrow} 4)$: the pair of $((p_k \to p_j)!e_k, (p_j \leftarrow p_k)?v'_j)$.

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 $s^1_{p_i}$, $s^2_{p_i}$, ..., $s^{m_1}_{p_i}$, and $s^1_{p_j}$, $s^2_{p_j}$, ..., $s^{m_1}_{p_j}$, and $s^1_{p_k}$, $s^2_{p_k}$, ..., $s^{m_1}_{p_k}$, 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.

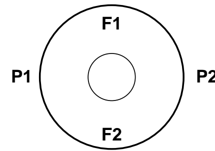

Fig. 13.30. Two dining philosophers table with forks

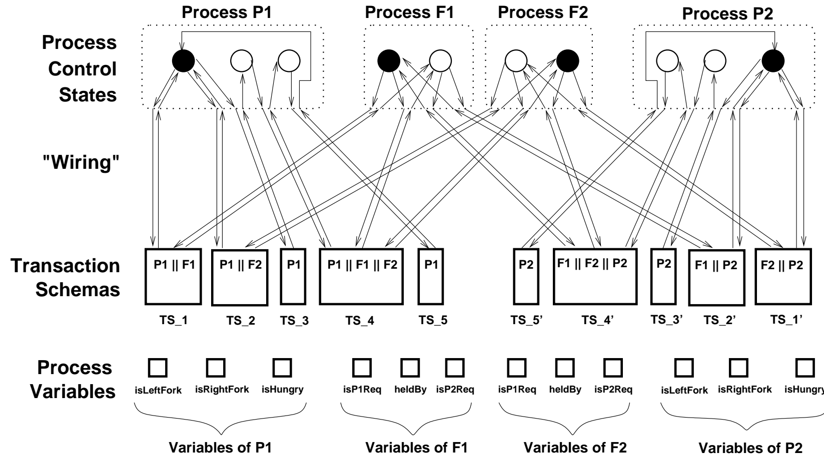

Fig. 13.31. Two dining philosophers CTP program

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.

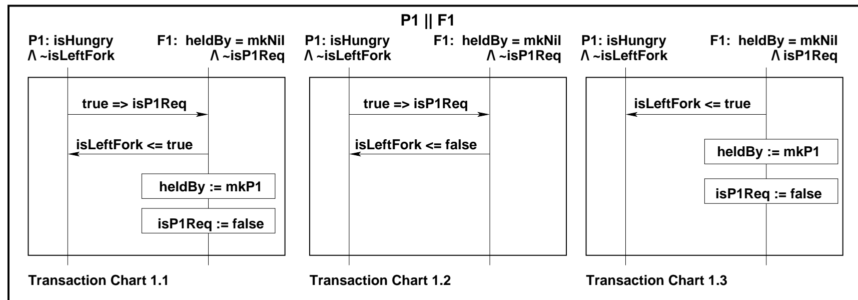

Fig. 13.32. Transaction schema TS_1

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.

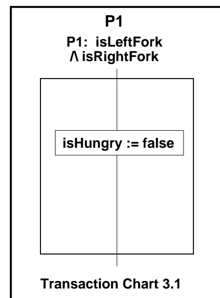

Fig. 13.33. Transaction schema TS_3

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.

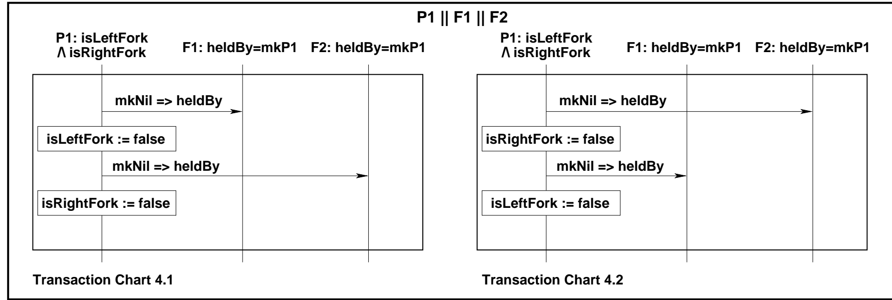

Fig. 13.34. Transaction schema TS_4

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 $P2$.

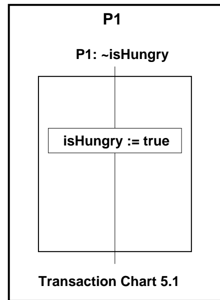

Fig. 13.35. Transaction schema TS_5

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
$P\Psi = P \xrightarrow[m]{} \Psi$
$\Psi = \Pi \times \Sigma \times \Theta$

Annotation:

$P\Psi$: 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)
$\Theta = \text{Pos} \xrightarrow[m]{} \text{VAL}$
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
  $P\Delta = P \ \overrightarrow{m} \ \Delta$

Annotation:

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

type
  $\Delta = T \times Chtn \times \Phi$
  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).