STUDY OF GRAPHICAL AND TEMPORAL SPECIFICATION TECHNIQUES

Christian Krog Madsen
c973746

June 2, 2003

Computer Science and Technology
Informatics and Mathematical Modelling
Technical University of Denmark

Contents

1 Introduction

It is broadly recognized that succesful software engineering relies on succesful domain and requirements engineering. For this reason many techniques have been developed for describing domains and specifying requirements. Some of these techniques are based on various forms of structured natural language, some are based on diagrams and some are based on formal mathematical and logical specification languages.

For most software engineering projects is is wise to use a mixture of such techniques, because each technique has its advantages and disadvantages. Natural language is readily understandable by the client. Graphical notations are generally good at providing an overview of a large system. Formal languages admit verification using formal proofs.

The problem with using several techniques is that they do not have a common semantics, so it is not possible to formally relate parts of a system specified in one notation to parts specified in another notation.

The aim of this report is to study three graphical specification techniques and attempt to compare them against a formal specification language. The three notations are those of Petri Nets, Message Sequence Charts and Statecharts. In the first part of the report each notation is first described in natural language supported by example diagrams, then the syntax and static semantics of the notations are formalised in the specification language RSL. The second part of the report contains three larger examples, which are modelled separately in a graphical notation and RSL. The two models are aimed to be as similar as possible with the very different properties of the specification notations.

The first example deals with the control part of the MIPS R10000 superscalar microprocessor. This example is modelled using a Petri Net.

The second example specifies the exchange of messages between a station and an access point in an IEEE 802.11 wireless network using Message Sequence Charts. The presentation gives just a high level overview of IEEE 802.11 and ignores many details such as message loss and power saving functions.

The last example models a small home wireless rain gauge using Statecharts. The reactive nature of such a device is well suited for the state and event paradigm in Statecharts.

In Section 2 three variants of Petri Nets are described and formalised. These are Condition-Event Nets, Place-Transition Nets and Coloured Petri Nets. In Section 3 basic and high level Message Sequence Charts are described and formalised. In Section 4 Statecharts are described and formalised. Section 5 contains the three examples discussed above. Finally, Section 6 offers a conclusion.

2 Petri Nets

In this section we review several variants of Petri Nets, ranging from the basic Condition-Event Nets to Coloured Petri Nets. Each of the discussed types of Petri Nets are modelled formally in RSL. Petri Nets were first described by Carl Adam Petri in his doctoral thesis [29] in 1962.

Petri Nets are composed of states, transitions and arrows linking states to transitions and transitions to states. Depending on the type of Petri Net states may be called places or conditions, while transitions are also referred to as events.

The description of Condition-Event Nets and Place-Transition Nets is based on Reisig [31]. The description of Coloured Petri Nets is based on Jensen [16].

In the sequel we shall avail outselves to a somewhat imprecise use of language for readability. When some abstract entity has a graphical representation we shall use the name of the abstract entity to also denote its graphical representation. For example, in a Petri Net a state is usually represented graphically as a circle, while a transition is represented by a rectangle. Suppose an arrow extends from the perimeter of the circle to the perimeter of the rectangle. Then, we shall say that the arrow links the state to the transition. Really, what we should say is that the arrow links the graphical representation of the state to the graphical representation of the transition.

2.1 Condition-Event Nets

2.1.1 Description

Condition-Event Nets (CEN) are the most basic type of Petri Net. A CEN consists of *conditions*, *events* and links from conditions to events and from events to conditions. Conditions are drawn as circles, while events are drawn as rectangles. An event may have a set of *preconditions* which are conditions. Similarly, an event may have a set of *postconditions* which are also conditions. A precondition of an event is represented graphically by an arrow emanating from the precondition and ending at the event. Similarly, a postcondition is represented by an arrow emanating from the event and ending at the postcondition.

A condition may be *marked* with a *token*. Graphically this is represented by drawing a disc inside the condition. A *marking* of a CEN is an assignment of tokens to some of the conditions in the CEN.

A condition that is marked with a token is said to be *fulfilled*. Conversely, a condition that is not marked is said to be *unfilfilled*. If all the preconditions of an event are fulfilled and all the postconditions of the event are unfulfilled, the event is said to be *activated*. An event that is activated may *occur*. If an event occurs, all its preconditions become unfulfilled and all its postconditions become fulfilled. Figure 1 illustrates the occurrence of an event.

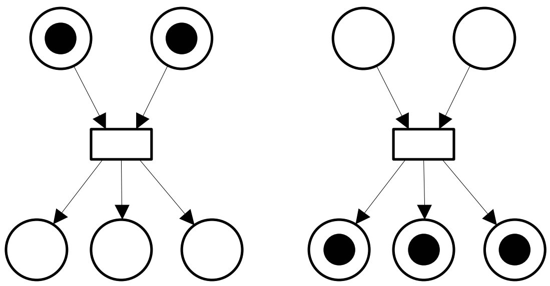

Figure 1: Occurrence of an event in a Condition-Event Net. The left net shows the marking before the occurrence, the right net shows the marking after the occurrence.

Example 1. Figure 2 illustrates a CEN with a marking. The net represents a simplified model of the classical Dining Philosphers problem posed by Dijkstra [6].

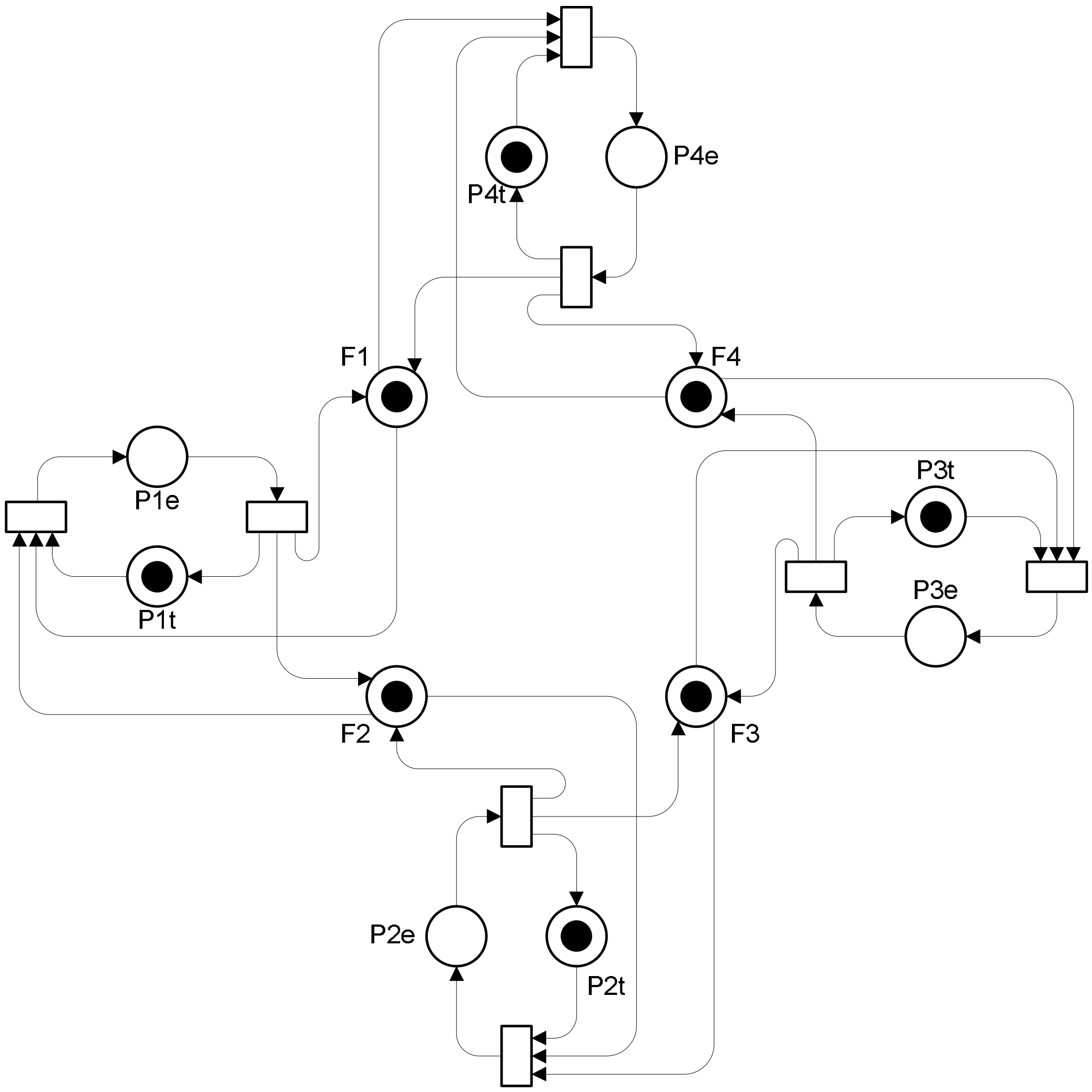

Figure 2: Example Condition-Event Net modelling the Dining Philosophers problem with four philosophers.

The problem is set in a monastery where five philosophers spend their life engaged in thinking. Their thinking is only interrupted when they have to eat. The monastery has a circular dining table with a place for each of the philosophers. At the center of the table is a bowl with an endless supply of spaghetti. On the table there is a plate for each place and a fork between each pair of adjacent plates. To eat, a philosopher must use the two forks adjacent to his plate. The problem is then to devise a strategy that will allow the philosophers to eat without risking starvation.

In the CEN there are only four philosophers, each of which is represented by two conditions, labelled Pxt and Pxe, where x is the number of the philosopher. When Pxt is marked, philosopher x is thinking. When Pxe is marked, philosopher x is eating. The final four conditions, Fx, represent the four forks. When Fx is marked, fork x is free.

In order for philosopher x to begin eating, he must currently be thinking, and the two adjacent forks must be free. This is represented by an event with preconditions Pxt, Fx and $F(x + 1 \bmod 4)$. While philosopher x is eating he cannot be thinking, and the two adjacent forks are not free. This is represented by letting the postcondition of the event be Pxe.

When philosopher x stops eating, he places the two forks on the table and begins thinking. This is represented by an event with precondition Pxe and postconditions Pxt, Fx and $F(x + 1 \bmod 4)$. □

2.1.2 Formalisation

We now proceed to formalise the above narrative description of CENs. We first specify a static CEN.

scheme ConditionEventNet =
  class
    type
      CEN = {| c : CEN′ • wf_CEN(c) |},
      CEN′ = Cond-**set** × Event-**set** × PreCond × PostCond × Marking,
      Cond,
      Event,
      PreCond = Event $\overrightarrow{m}$ Cond-**set**,
      PostCond = Event $\overrightarrow{m}$ Cond-**set**,
      Marking = Cond $\overrightarrow{m}$ Mark,
      Mark == Empty | Token

    value
      wf_CEN : CEN′ → **Bool**
      wf_CEN(cs, es, precs, postcs, mark) ≡
        /∗ 1 ∗/
        dom precs = es ∧
        /∗ 2 ∗/
        dom postcs = es ∧
        /∗ 3 ∗/
        {c |
         c : Cond •
           ∃ cs : Cond-**set** •
             c ∈ cs ∧ cs ∈ **rng** precs ∪ **rng** postcs} = cs ∧
        /∗ 4 ∗/
        (∀ e : Event • e ∈ es ⇒ precs(e) ∪ postcs(e) ≠ {}) ∧
        /∗ 5 ∗/
        dom mark = cs
  end

Annotations

- A Condition-Event Petri Net (CEN) consists of a set of conditions, a set of events, preconditions, postconditions and a marking.
- Only well-formed CENs will be considered.
- Conditions and Events are further unspecified entities.
- Preconditions are a mapping from Events to sets of Conditions.
- Postconditions are a mapping from Events to sets of Conditions.

- A Marking is an assignment of Mark to Conditions.
- A Mark is either Empty or a Token.
- A CEN is well-formed if
- (1) every Event in the set of Events is included in the maps of pre- and (2) post-conditions, and,
- (3) every Condition is a pre- or postcondition of some Event, and,
- (4) every Event has at least one pre- or postcondition, and,
- (5) the marking includes all Conditions.

□

Next, we describe the dynamic aspects of a CEN, namely what it means for a condition to be fulfilled or unfulfilled and what it means for an event to be activated and to occur.

context: ConditionEventNet

scheme ConditionEventNetDynamics =
  extend ConditionEventNet **with**
  class
    value
      fulfilled : Cond × CEN $\xrightarrow{\sim}$ **Bool**
      fulfilled(cond, (cs, es, precs, postcs, mark)) ≡ mark(cond) = Token
      pre cond ∈ cs,

      unfulfilled : Cond × CEN $\xrightarrow{\sim}$ **Bool**
      unfulfilled(cond, (cs, es, precs, postcs, mark)) ≡ mark(cond) = Empty
      pre cond ∈ cs,

      activated : Event × CEN $\xrightarrow{\sim}$ **Bool**
      activated(evt, cen) ≡
        let (cs, es, precs, postcs, mark) = cen **in**
          (∀ c : Cond • c ∈ precs(evt) ⇒ fulfilled(c, cen)) ∧
          (∀ c : Cond • c ∈ postcs(evt) ⇒ unfulfilled(c, cen))
        end
      pre let (cs, es, precs, postcs, mark) = cen **in** evt ∈ es **end**,

      occur : Event × CEN $\xrightarrow{\sim}$ CEN
      occur(evt, cen) ≡
        let (cs, es, precs, postcs, mark) = cen **in**
          (cs, es, precs, postcs,
           mark † [c ↦ Empty | c : Cond • c ∈ precs(evt)] †
           [c ↦ Token | c : Cond • c ∈ postcs(evt)])
        end
      pre activated(evt, cen)
  end

Annotations

- A Condition is fulfilled in a CEN if the marking assigns a token to that Condition.

- A Condition is unfulfilled if the marking assigns empty to that Condition.
- An Event is activated if all its preconditions are fulfilled and all its postconditions are unfulfilled.
- The occurrence of an activated event gives a new CEN where all preconditions of the event are unfulfilled and all postconditions of the event are fulfilled.

□

2.2 Place-Transition Nets

2.2.1 Description

A simple extension to the Condition-Event nets is to allow a marking to assign more than one token to a condition. The extended nets are known as Place-Transition Nets (PTN). Conditions are now called *places* and events are called *transitions*.

In a PTN the places are labelled with a positive integer called the *capacity*. This indicates the maximum number of tokens that may be assigned to that place. The capacity may be omitted, which is interpreted as unlimited capacity. Additionally, arrows are labelled with a positive integer called the *weight*. If an arrow from a place, P, to a transition, T, is labelled with x, this signifies that for T to be activated, there must be at least x tokens at P, and when T occurs, x tokens will be removed from P. If an arrow from a transition, T, to a place, P, is labelled with x, this signifies that for T to be activated, x added to the number of tokens at P must be at most equal to the capacity of P, and if T occurs, x tokens will be added to the marking of P. If an arrow is not labelled it is to be understood as an implicit labelling with 1. Figure 3 shows the occurrence of a transition in a PTN.

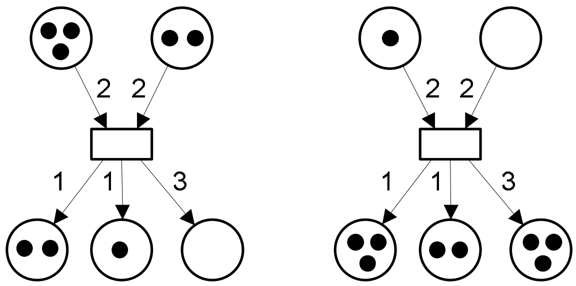

Figure 3: Occurrence of a transition in a Place-Transition Net. The left net shows the marking before the occurrence, the right net shows the marking after the occurrence.

Example 2. Figure 4 shows an example PTN modelling four processes that access a common critical resource. One process writes to the resource, while the other three processes read from the resource. To ensure data integrity, mutual exclusion must be enforced between the writing process and the reading processes.

The protocol for mutual exclusion requires a reading process to claim a key before it may read, while the writing process is required to claim three keys before it may write. A process that cannot get the required number of keys must wait until more keys become available. The place *Keys* holds a token for each key that is unused. When a process finishes reading or writing it returns the claimed keys to the place *Keys* and proceeds to do some processing that does not access the critical resource.

□

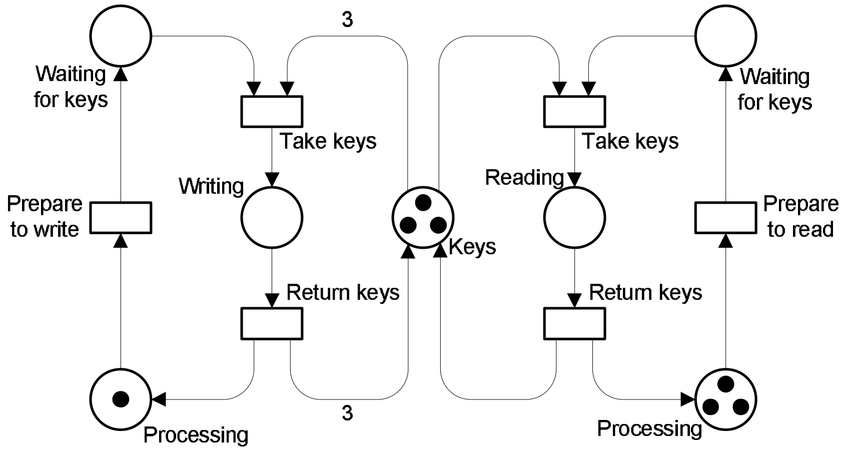

Figure 4: Example Place-Transition Net

2.2.2 Formalisation

We first formalise the static PTN.

scheme PlaceTransitionNet =
  class
    type
      PTN = {| ptn : PTN′ • wf_PTN(ptn) |},
      PTN′ = (Place $\underset{m}{\to}$ **Nat**) × Trans-**set** × Preset × Postset × Marking,
      Place,
      Trans,
      Preset = Trans $\underset{m}{\to}$ (Place × PosNat)-**set**,
      Postset = Trans $\underset{m}{\to}$ (Place × PosNat)-**set**,
      Marking = Place $\underset{m}{\to}$ Mark,
      Mark = PosNat,
      PosNat = {| n : **Nat** • n > 0 |}

    value
      wf_PTN : PTN′ → **Bool**
      wf_PTN(ps, ts, pres, posts, mark) ≡
        /∗ 1 ∗/
        dom pres = ts ∧
        /∗ 2 ∗/
        dom posts = ts ∧
        /∗ 3 ∗/
        {p |
         p : Place •
          ∃ pns : (Place × **Nat**)-**set**, n : **Nat** •
            (p, n) ∈ pns ∧ pns ∈ **rng** pres ∪ **rng** posts} ⊆ **dom** ps ∧
        /∗ 4 ∗/
        (∀ t : Trans • t ∈ ts ⇒ pres(t) ∪ posts(t) ≠ {}) ∧
        /∗ 5 ∗/
        (∀ t : Trans •
          ∼(∃ n1, n2 : **Nat**, p : Place •
            n1 ≠ n2 ∧ p ∈ **dom** ps ∧
            ({(p, n1), (p, n2)} ⊆ pres(t) ∨
             {(p, n1), (p, n2)} ⊆ posts(t)))) ∧

/* 6 */
dom mark = **dom** ps $\wedge$
/* 7 */
($\forall$ p : Place $\bullet$ p $\in$ **dom** ps $\Rightarrow$ mark(p) $\leq$ ps(p))
end

Annotations

- A Place Transition Net consists of a set of Places with associated capacities, a set of Transitions, a Preset, a Postset and a Marking;
- Only well-formed PTNs will be considered;
- Places and Transitions are further unspecified entities;
- Presets are a mapping from Transitions to sets of pairs of places and weights;
- Postsets are a mapping from Transitions to sets of pairs of places and weights;
- A Marking is a mapping of Places to Marks;
- A Mark is a positive integer;
- A PTN is well-formed if
- (1) every Transition in the set of Transitions is included in the domain of the maps of Presets and (2) Postsets, and,
- (3) every Place is in the pre- or postset of some Transition, and,
- (4) every transition has a non-empty preset or postset, and,
- (5) no Transition can have a preset or postset that includes the same Place more than once with different weights, and,
- (6) the Marking covers all Places, and,
- (7) for every Place the number of tokens assigned to it in the Marking must be at most equal to the capacity of the Place.

$\square$

Now, we formalise the dynamic aspects of PTN, namely what it means for a transition to be *activated* and for a transition to *occur*. Note, unlike for CENs there is no notion of a place being fulfilled or unfulfilled.

context: PlaceTransitionNet

scheme PlaceTransitionNetDynamics =
  extend PlaceTransitionNet **with**
  class
    value
      activated : Trans $\times$ PTN $\xrightarrow{\sim}$ **Bool**
      activated(t, ptn) $\equiv$
        let (ps, ts, pres, posts, mark) = ptn **in**
          ($\forall$ p : Place, n : **Nat** $\bullet$ (p, n) $\in$ pres(t) $\Rightarrow$ mark(p) $\geq$ n) $\wedge$
          ($\forall$ p : Place, n : **Nat** $\bullet$
            (p, n) $\in$ posts(t) $\Rightarrow$ mark(p) + n $\leq$ ps(p))

end
pre let (ps, ts, pres, posts, mark) = ptn **in** t ∈ ts **end**,

occur : Trans × PTN $\stackrel{\sim}{\to}$ PTN
occur(t, ptn) ≡
let (ps, ts, pres, posts, mark) = ptn **in**
(ps, ts, pres, posts,
mark †
[p ↦ mark(p) − n | p : Place, n : **Nat** • (p, n) ∈ pres(t)] †
[p ↦ mark(p) + n | p : Place, n : **Nat** • (p, n) ∈ posts(t)])
end
pre activated(t, ptn)
end

Annotations

- A Transition is activated if for every place in its preset there are at least as many tokens as the weight of the corresponding arrow, and if for every place in its postset the number of tokens at that place added to the weight of the corresponding arrow is at most equal to the capacity of the place.
- The occurrence of an activated transition produces a new marking in which the number of tokens at each of the places in the preset is reduced by the weight of the corresponding arrow, and in which the number of tokens at each of the places in the postset is increased by the weight of the corresponding arrow.

□

2.3 Coloured Petri Nets

2.3.1 Description

In the Petri Nets variants described above, tokens are indistinguishable, i.e. there is no way to tell one token apart from another. In this section we discuss Coloured Petri Nets (CPN), which is an extension of PTNs where a type-value system is introduced for tokens. The term coloured refers to the fact that tokens are now distinguishable in that they have a value, called their *colour*, which is of a particular type, called their *colour set*. A colour set may define both simple and composite values.

In a CPN each place has an associated colour set specifying the colour set of tokens at that place. The marking of a place is a multi-set[1] over the colour set of the place. A transition may have a sequence of *guard* expressions which evaluate to a Boolean value. Arrows from places to transitions and from transitions to places are called arcs. Arcs are inscribed with expressions that evaluate to a multiset over the colour set associated with the place from which they emanate or terminate.

Expressions may contain *variables*. A variable is typed with a colour set and may be bound to any value in that colour set. A *binding* is an assignment of colours to variables. A *binding element* is a pair (t, b) of a transition, t, and a binding, b, where b assigns a colour to each variable that appears in an arc expression of an arc emanating from or terminating at t.

[1] A multi-set is an unordered collection of values, in which the same value may appear more than once. Multi-sets are also known as bags. The notation $1`a + 2`b + 4`c$ denotes a multi-set containing one $a$ value, two $b$ values and four $c$ values. If the number and reverse prime symbol are omitted, it is interpreted as a single value, e.g. $a + 2`b$ is equivalent to $1`a + 2`b$.

For a given binding a transition is *enabled* if the conjunction of its guard expressions evaluates to true and for each arc terminating at the transition, the multiset value of the arc expression may be removed from the place at which the arc emanates.

A transition that is enabled under a given binding may *occur*. In that case tokens are removed from its input places and tokens are added to its output places. The colour of the tokens are determined by the value of the corresponding arc expressions.

Complex CPNs may be simplified by splitting them into several smaller nets organised in a hierarchy. The simple nets are called *pages*. A page may have several *instances* which differ only in that each has its own marking, which is independent of the markings of the other instances of the page. A transition in a net may be elaborated as a page, such that the page specifies the detailed behaviour of the transition. In this case the transition is labelled with the letters *HS*. It is important to realise that a hierarchical net can always be converted to a non-hierarchical net specifying the same behaviour. Therefore, allowing hierarchical nets does not add to the expressibility of CPNs, but it does improves readability.

The definition of CPNs does not mandate a particular language for declarations. The most often used language for specifying colours, colour sets, functions and expressions is known as CPN ML. This language is an extension of Standard ML [27]. Here, we briefly list the additional facilities and assume the reader is familiar with Standard ML. Refer to [4] for a thorough reference to CPN ML.

Table 1 lists the facilities available in CPN ML for declaring colour sets. A range of built-in operators are available for the simple colour sets derived from *bool*, *int*, *real* and *string*. These operators include logical operators for *bool*, arithmetic operators for *int* and *real*, standard trigonometric, logarithmic and exponential functions for *real* and concatenation, substring and conversion functions for *string*.

Multi-sets play an important role in CPNs, so CPN ML has operators for constructing, manipulating and comparing multi-sets over colour sets. Multi-sets are constructed using the back-quote operator (\`) and the multi-set union operator (+). The value *empty* denotes the empty multi-set.

CPN ML supports typed variables, which are local to a transition. Reference variables with global, page or instance level scope are also supported.

Example 3. We illustrate CPN by revisiting the Dining Philosophers example from Section 2.1. This time we have five dining philosophers. The CPN diagram is shown in Figure 5. Because of the expresssibility of CPN we need only 3 places compared with the 12 conditions in the CEN model. Before, we had a condition for each fork. Now, there is only one place which can be marked with a colour to indicate which of the forks are free. Similarly, the two conditions for each philosopher in the first example are translated into two shared places which are marked with a colour to indicate which philosophers are eating and which are thinking. The function $S(x)$ is introduced to provide the mapping from a philosopher to the forks he uses.

□

2.3.2 Formalisation

The above narrative description of CPNs is now formalised in RSL. First, the static CPN is specified. The net inscriptions (i.e. colour set declarations, colour definitions, arc expressions and transition guards) are abstracted as sorts to avoid having to define the full syntax of CPN ML or some other inscription language.

scheme ColouredPetriNet =

CPN ML colour set declaration	RSL Equivalent
Simple Colour Sets	
color A = **unit**	**type** A = **Unit**
color B = **bool**	**type** B = **Bool**
color C = **int**	**type** C = **Int**
color D = **real**	**type** D = **Real**
color E = **string**	**type** E = **Text**
color F = **unit with** e	**type** F == e
color G = **bool with** (no, yes)	**type** G == no | yes †
color H = **int with** 10..40	**type** H = {| i : **Int** • i $\in$ { 10..40 } |}
color I = **real with** 0.5..1.5	**type** I = {| r : **Real** • r $\geq$ 0.5 $\wedge$ r $\leq$ 1.5 |}
color J = **string with** "a".."z"	**type** J = {| t : **Text** • $\forall$ i:**Nat** • i $\in$ **inds** t $\Rightarrow$ t(i) $\geq$ 'a' $\wedge$ t(i) $\leq$ 'z' |}
color K = **string with** "a".."z" and 2..4	**type** K = {| t : **Text** • **len** t $\in$ {2..4} $\wedge$ $\forall$ i:**Nat** • i $\in$ **inds** t $\Rightarrow$ t(i) $\geq$ 'a' $\wedge$ t(i) $\leq$ 'z' |}
color L = **with** red | green | blue | yellow	**type** L == red | green | blue | yellow
color M = **index** list **with** 2..6	**type** M == list2 | list3 | list4 | list5 | list6 ⋆
Compound Colour Sets	
color N = **product** A * B * C	**type** N = A $\times$ B $\times$ C
color O = **record** a:A * b:B * c:C	**type** O :: a : A b : B c : C
color P = **list** A	**type** P = A*
color Q = **list** A **with** 2..5	**type** Q = {| a : A* • **len** a $\geq$ 2 $\wedge$ **len** a $\leq$ 5 |}
color R = **union** a + b:B + c:C	**type** R1 == a, R2 == b(B), R3 == c(C), R = R1 | R2 | R3
color S = **subset** C **by** Even	**type** S = {| c : C • even(c) |}
color T = **subset** C **with** [1, 2, 4, 8, 16]	**type** T = {| c : C • c $\in$ {1, 2, 4, 8, 16} |}

†: The only effect of this colour set is to rename *false* to *no* and *true* to *yes*. Standard ML Boolean operators such as *not*, *andalso*, *orelse* may be applied to *no* and *yes* just as to *false* and *true*.

⋆: There is no proper RSL equivalent for this colour set. A value definition is probably the closest match: **type** M1 = list2 | list3 | list4 | list5 | list6 **value** M = [2 $\mapsto$ list2, 3 $\mapsto$ list3, 4 $\mapsto$ list4, 5 $\mapsto$ list5, 6 $\mapsto$ list6]

Table 1: CPN ML colour set declarations.

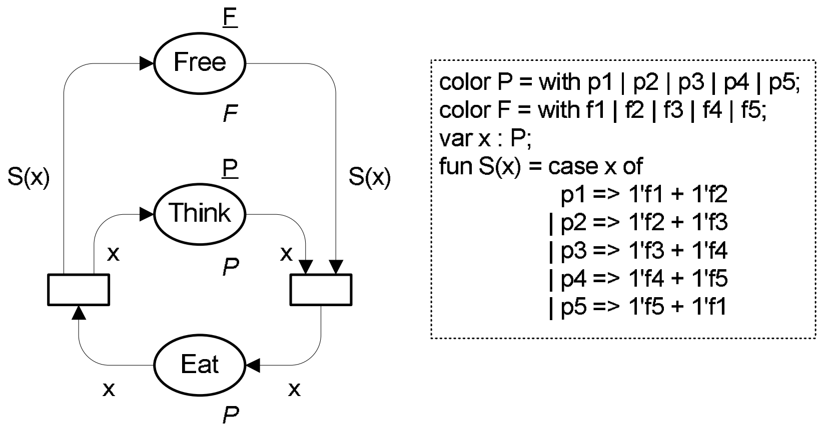

Figure 5: CPN model of the Dining Philosophers problem.

class
  type
    $\mathrm{CPN} = \{|\ \mathrm{cpn} : \mathrm{CPN}' \bullet \mathrm{wf\_CPN}(\mathrm{cpn})\ |\}$,
    $\mathrm{CPN}' = \mathrm{ColFun} \times \mathrm{Guard} \times \mathrm{Preset} \times \mathrm{Postset} \times \mathrm{Marking}$,
    $\Sigma = \sigma\textbf{-infset}$,
    σ,
    $\mathrm{ColFun} = \mathrm{Place} \xrightarrow[m]{} \Sigma$,
    $\mathrm{Guard} = \mathrm{Trans} \xrightarrow[m]{} \mathrm{Pred}^*$,
    Place,
    Trans,
    $\mathrm{Preset} = \mathrm{Trans} \xrightarrow[m]{} ((\mathrm{Place} \times \mathrm{Exp}) \xrightarrow[m]{} \mathbf{Nat})$,
    $\mathrm{Postset} = \mathrm{Trans} \xrightarrow[m]{} ((\mathrm{Place} \times \mathrm{Exp}) \xrightarrow[m]{} \mathbf{Nat})$,
    $\mathrm{Marking} = \mathrm{Place} \xrightarrow[m]{} (\sigma \xrightarrow[m]{} \mathbf{Nat})$,
    $\mathrm{Binding} = \mathrm{Var} \xrightarrow[m]{} (\sigma \xrightarrow[m]{} \mathbf{Nat})$,
    Var,
    Exp,
    Pred

  value
    $\mathrm{wf\_CPN} : \mathrm{CPN}' \to \mathbf{Bool}$
    $\mathrm{wf\_CPN}(\mathrm{cf, g, pres, posts, mark}) \equiv$
      $/*\ 1\ */$
      $\mathbf{dom}\ \mathrm{pres} = \mathbf{dom}\ \mathrm{g} \wedge$
      $/*\ 2\ */$
      $\mathbf{dom}\ \mathrm{posts} = \mathbf{dom}\ \mathrm{g} \wedge$
      $/*\ 3\ */$
      $\{\mathrm{p}\ |$
       $\mathrm{p} : \mathrm{Place} \bullet$
         $\exists\ \mathrm{e} : \mathrm{Exp},\ \mathrm{t} : \mathrm{Trans} \bullet$
           $(\mathrm{p, e}) \in \mathbf{dom}\ \mathrm{pres(t)} \cup \mathbf{dom}\ \mathrm{posts(t)}\} \subseteq \mathbf{dom}\ \mathrm{cf} \wedge$
      $/*\ 4\ */$
      $(\forall\ \mathrm{t} : \mathrm{Trans} \bullet \mathbf{dom}\ \mathrm{pres(t)} \neq \{\} \wedge \mathbf{dom}\ \mathrm{posts(t)} \neq \{\}) \wedge$
      $/*\ 5\ */$
      $\mathbf{dom}\ \mathrm{mark} = \mathbf{dom}\ \mathrm{cf} \wedge$
      $/*\ 6\ */$
      $(\forall\ \mathrm{p} : \mathrm{Place} \bullet$

$$
\begin{array}{l}
\quad\quad\quad p \in \textbf{dom}\ cf \Rightarrow \\
\quad\quad\quad\quad (\forall\ c : \sigma \bullet c \in \textbf{dom}\ mark(p) \Rightarrow typeof(c) = cf(p))) \wedge \\
\quad\quad /{*}\ 7\ {*}/ \\
\quad\quad (\forall\ t : Trans \bullet \\
\quad\quad\quad t \in \textbf{dom}\ pres \cup \textbf{dom}\ posts \Rightarrow \\
\quad\quad\quad\quad (\forall\ p : Place,\ e : Exp \bullet \\
\quad\quad\quad\quad\quad\quad (p, e) \in \textbf{dom}\ pres(t) \cup \textbf{dom}\ posts(t) \Rightarrow typeof(e) = cf(p))),
\end{array}
$$

$$
\begin{array}{l}
eval : Exp \times Binding \stackrel{\sim}{\rightarrow} \sigma \underset{m}{\rightarrow} \textbf{Nat}, \\
evalP : Pred \times Binding \stackrel{\sim}{\rightarrow} \textbf{Bool}, \\
\\
evalPl : Pred^{*} \times Binding \stackrel{\sim}{\rightarrow} \textbf{Bool} \\
evalPl(pl, b) \equiv evalP(\textbf{hd}\ pl, b) \wedge evalPl(\textbf{tl}\ pl, b) \\
\textbf{pre dom}\ b = obs\_var(pl), \\
\\
typeof : Exp \rightarrow \Sigma, \\
typeof : \sigma \rightarrow \Sigma, \\
\\
typematch : (\sigma \underset{m}{\rightarrow} \textbf{Nat}) \rightarrow \textbf{Bool} \\
typematch(ms) \equiv \\
\quad (\forall\ c, c' : \sigma \bullet \{c, c'\} \subseteq \textbf{dom}\ ms \Rightarrow typeof(c) = typeof(c')), \\
\\
typeof : (\sigma \underset{m}{\rightarrow} \textbf{Nat}) \stackrel{\sim}{\rightarrow} \Sigma \\
typeof(ms) \equiv \textbf{let}\ c : \sigma \bullet c \in \textbf{dom}\ ms\ \textbf{in}\ typeof(c)\ \textbf{end} \\
\textbf{pre}\ typematch(ms), \\
\\
obs\_var : Exp \rightarrow Var\textbf{-set}, \\
obs\_var : Pred \rightarrow Var\textbf{-set}, \\
\\
obs\_var : Pred^{*} \rightarrow Var\textbf{-set} \\
obs\_var(pl) \equiv \\
\quad \{v \mid v : Var,\ p : Pred \bullet p \in \textbf{elems}\ pl \wedge v \in obs\_var(p)\}
\end{array}
$$

end

Annotations

- A CPN consists of a colour function, a set of guards, presets, postsets and a marking.
- A colour set (Σ) is a possibly infinite set of colours.
- A colour (σ) is a further undefined entity.
- The colour function maps places to colour sets.
- For each transition there is a a guard, which is a possibly empty sequence of predicates.
- Places and transitions are further undefined entities.
- Each transition has a preset, which is a multi-set of pairs of places and expressions.
- Each transition has a postset, which is a multi-set of pairs of places and expressions.
- A marking assigns a multi-set of colours to each place.

- A binding maps variables to multi-sets of colours.
- Variables, expressions and predicates are further undefined entities.
- A CPN is wellformed, if
- (1) the set of transitions which have a preset is identical to the set of transitions which have a guard, and;
- (2) the set of transitions which have a postset is identical to the set of transitions which have a guard, and;
- (3) the set of places which are in the preset or postset of some transition is a subset of those places which are associated with a colour set, and;
- (4) no transition has an empty preset or postset, and;
- (5) every place which is associated with a colour set also has a marking, and;
- (6) the marking of every place consists of a multi-set over the colour set associated with the place, and;
- (7) for every transition every arcs emanating from or terminating at the transition is inscribed with an expression, which has a colour set equal to the colour set associated with the place of the arc.
- There is a function which evaluates an expression under a given binding to a multi-set over some colour set.
- There is a function which evaluates a predicate under a given binding to a Boolean value.
- A predicate list is evaluated as the conjunction of the values of the member predicates.
- It is possible to observe the colour set (type) of an expression.
- It is possible to observe the colour set (type) from a colour.
- A multi-set of colours has matching types if any two colours in the multi-set have the same type.
- It is possible to observe the colour set from a multi-set of colours, if the multi-set has matching types.
- The variables of an expression, predicate or predicate list can be observed.

$\square$

Before we turn to the dynamic aspects of CPNs, we specify four operations on multi-sets over colour sets: *union*, *distributed union*, *difference* and *subset*.

context: ColouredPetriNet

scheme ColourMultiSet =
  extend ColouredPetriNet **with**
  class
    value
      ms_union : $(\sigma \underset{m}{\rightarrow} \mathbf{Nat}) \times (\sigma \underset{m}{\rightarrow} \mathbf{Nat}) \rightarrow (\sigma \underset{m}{\rightarrow} \mathbf{Nat})$

```
      ms_union(msa, msb) ≡
        msa \ dom msb ∪ msb \ dom msa ∪
        [c ↦ msa(c) + msb(c) | c : σ • c ∈ dom msa ∩ dom msb],

      ms_dunion : (σ -m-> Nat)-set → (σ -m-> Nat)
      ms_dunion(mss) ≡
        if mss = {} then [ ]
        else
          let ms : (σ -m-> Nat) • ms ∈ mss in
            ms_union(ms, ms_dunion(mss \ {ms}))
          end
        end,

      ms_diff : (σ -m-> Nat) × (σ -m-> Nat) → (σ -m-> Nat)
      ms_diff(msr, msa) ≡
        msa \ dom msr ∪
        [c ↦ if msa(c) − msr(c) ≥ 0 then msa(c) − msr(c) else 0 end |
         c : σ • c ∈ dom msr ∩ dom msa],

      ms_subset : (σ -m-> Nat) × (σ -m-> Nat) → Bool
      ms_subset(msr, msa) ≡
        (∀ c : σ • c ∈ dom msr ⇒ c ∈ dom msa ∧ msa(c) ≥ msr(c))
  end
```

Annotations

- The union of two multi-sets is obtained as the union of those elements which are in only one of the multi-sets with the sum of those elements which are in both multi-sets.
- The distributed union of a set of multi-sets is defined recursively as the union of one member of the set with the distributed union of the rest of the set. The distributed union of the empty set is the empty multi-set.
- The difference between two multi-sets is obtained by removing the elements of the first multi-set from the second multi-set. If the second multi-set does not contain as many elements as should be removed, all elements are removed.
- A multi-set is a subset of another multi-set, if and only if for every element in the first multi-set, the second multi-set contains at least as many instances of that element as the first multi-set.

□

With the above specification of multi-sets of colours we are now ready to specify what it means for a transition to be *enabled* and to *occur* in a CPN.

context: ColouredPetriNet, ColourMultiSet

```
scheme ColouredPetriNetDynamics =
  extend ColourMultiSet with
  class
    value
      enabled : Trans × Binding × CPN -~-> Bool
      enabled(t, b, cpn) ≡
```

```
    let (cf, gu, pres, posts, mark) = cpn in
      evalPl(gu(t), b) ∧
      (∀ p : Place •
          let ms = {eval(e, b) | e : Exp • (p, e) ∈ dom pres(t)} in
            ms_subset(ms_dunion(ms), mark(p))
          end)
    end
  pre
    let (cf, gu, pres, posts, mark) = cpn in
      t ∈ dom gu ∧
      dom b =
        obs_var(gu(t)) ∪
        {v |
         v : Var, p : Place, e : Exp •
           (p, e) ∈ pres(t) ∪ posts(t) ∧ v ∈ obs_var(e)}
    end,

  occur : Trans × Binding × CPN ~→ CPN
  occur(t, b, cpn) ≡
    let (cf, gu, pres, posts, mark) = cpn in
      (cf, gu, pres, posts,
       mark †
       [p ↦
         let ms = {eval(e, b) | e : Exp • (p, e) ∈ dom pres(t)} in
           ms_diff(ms_dunion(ms), mark(p))
         end | p : Place • ∃ e : Exp • (p, e) ∈ dom pres(t)] †
       [p ↦
         let ms = {eval(e, b) | e : Exp • (p, e) ∈ dom posts(t)} in
           ms_union(ms_dunion(ms), mark(p))
         end | p : Place • ∃ e : Exp • (p, e) ∈ dom posts(t)])
    end
  pre
    let (cf, gu, pres, posts, mark) = cpn in
      t ∈ dom gu ∧
      dom b =
        obs_var(gu(t)) ∪
        {v |
         v : Var, p : Place, e : Exp •
           (p, e) ∈ pres(t) ∪ posts(t) ∧ v ∈ obs_var(e)} ∧
      enabled(t, b, cpn)
    end
end
```

Annotations

- A transition is enabled under a given binding if its guard condition evaluates to true, and for every input place the value of the corresponding arc expression is a subset of the marking of the place.
- When an enabled transition occurs, tokens are removed from its input places and tokens are added to its output places, as determined by the value of the arc expressions under the given binding.

□

2.3.3 Timed Coloured Petri Nets

In the above description of CPNs we have neglected temporal aspects in CPNs. The CPN model of time is based on a global discrete or continuous clock. Discrete or continuous durations may be attached to transitions and arc expressions of arcs from transitions to places. Tokens may be labelled with a timestamp indicating the earliest time the token may be removed from a place.

To indicate that tokens of a particular colour set should have timestamps, the keywork **timed** is appended to the declaration of the colour set: **color** A = **product int** * **string timed**; A *timed multi-set* is a multi-set with timestamps: $2`(2, "monday")@[5, 16]+$ $3`(3, "tuesday")@[14, 20, 21]$.

Durations of transitions are specified as $@+x$ indicating that tokens added to the output places should be timestamped with the time of the global clock plus x, where x is an integer or real value. Durations may also be specified on output arcs by appending $@+x$ to the arc expressions.

In a Timed CPN a transition may occur if it is both *enabled* and *ready*, i.e. guards are fulfilled, the required tokens are available, and the timestamps of the tokens to be removed are less than or equal to the current model time.

The execution of a Timed CPN proceeds by executing all transitions that are enabled and ready. Whenever no further transitions are ready to be executed, the global clock is advanced to the next time at which one or more transitions are enabled and ready.

The model of CPNs given above could be refined to include a global clock, timestamps and durations, but we will not give such a model here.

3 Message Sequence Charts

In this section we describe Message Sequence Charts (MSC) which is a graphical notation for specifying sequences of message exchanges between entities. We describe the components of MSCs and then provide a formalisation of the syntax in RSL. We follow the syntax requirements defined by Reniers [33, 32].

Message Sequence Charts were first standardised by the CCITT (now ITU-T) as Recommendation Z.120 in 1992 [2]. The standard was later revised and extended in 1996 [14] and in 1999 [15]. 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. The most recent standard provides additional facilities for specifying the data that is passed in messages and also allows inline expressions.

3.1 Basic Message Sequence Charts

3.1.1 Description

A Basic Message Sequence Chart (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 handling 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.

Events specified on an instance are totally ordered in time. Two events cannot take place at the same time. Events on different instances are not ordered, except that message input by one instance must be preceded by the corresponding message output in another instance.

A *message input* represents the reception of a message from another instance or the environment. A *message output* represents the sending of a message to another instance or the environment. For each message output there must be a matching message input. A message exchange consists of a message output and a message input. A message exchange is represented as an arrow from the time line of the sending instance to the time line of the receiving instance. In case of messages exchanged with the environment, the side of the diagram can be considered to be the time line of the environment. The arrow is labelled with a message identifier.

Message exchange is asynchronous, i.e. message input is not necessarily simultaneous with message output.

Actions are events that are local to an instance. Actions are represented by a box on the time line with an action label inside.

Conditions describe a state that is common to a subset of instances in an MSC. Conditions have no semantic import and merely serve as documentation. Conditions are represented as a hexagon extending across the time lines of the instances for which the condition applies. The condition text is placed inside the hexagon.

Timers appear in two situations: either the setting of a timer and subsequent timeout, or the setting of a timer and subsequent reset. Timers are local to an instance. The setting of a timer is represented by a rectangle placed against the instance time line and labelled with a timer identifier. Timer reset is represented by an dashed arrow from the rectangle to a point on the time line below the rectangle. Timer timeout is represented by a fully drawn arrow from the rectangle to the time line below the rectangle.

An instance may create a new instance. This is called *process creation*. Conversely, an instance may cause itself to terminate, this is called *process termination*. Process creation is represented by a dashed arrow from the time line of the creating instance to a new instance symbol with associated time line. Process termination is represented by a cross, which is the last symbol on a time line.

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

Example 4. Figure 6 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 message exchanges with A. Note that B may either receive *Msg4* and then send *Msg5*, or, 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.

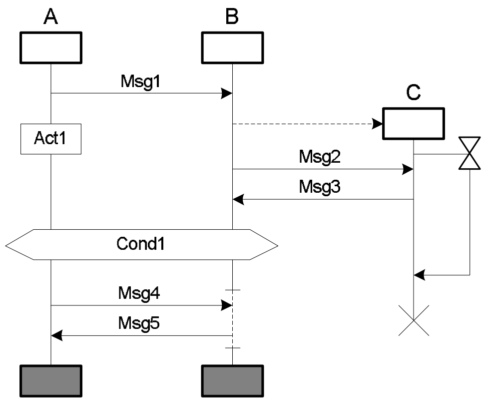

Figure 6: BMSC example.

□

3.1.2 Formalisation

We first formalise basic Message Sequence Charts. We defer the discussion of well-formedness conditions to Section 3.3.

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 basic Message Sequence Chart 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 coregion event.

- An action event has a name.
- A message event is either message input or 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 duration is either unspecified or has a name.
- Names are further unspecified entities.

□

3.2 High Level Message Sequence Charts

3.2.1 Description

We now extend the above definition of Basic Message Sequence Charts to allow several BMSC to form a 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 it connects, 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 (△) 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.

Example 5. Figure 7 shows 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 request a resource and the server responds by returning that resource. Finally, the client logs off and the server closes the connection.

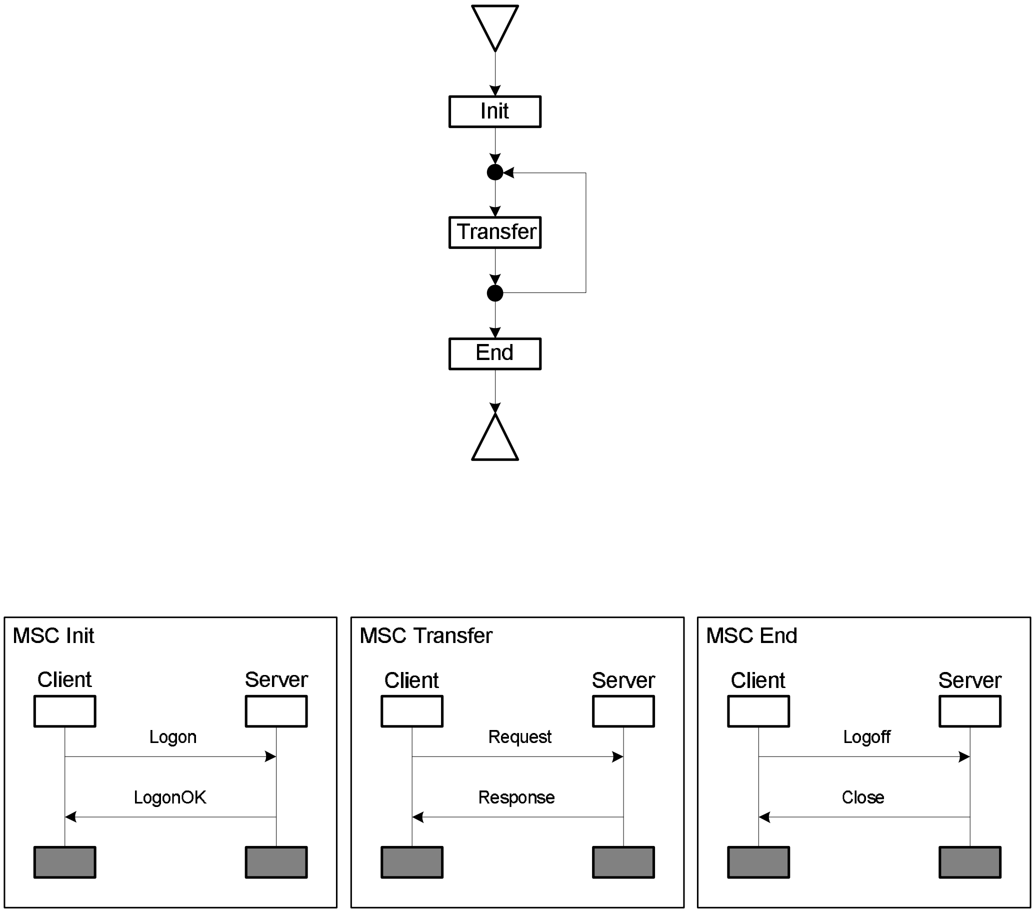

Figure 7: HMSC example.

□

3.2.2 Formalisation

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.

□

3.3 Well-formedness of Message Sequence Charts

Now that we have defined the full syntax of Message Sequence Charts we are ready to specify the requirements for a chart to be well-formed.

First, we specify conditions for a Basic Message Sequence Chart to be well-formed. These conditions were derived by Reniers [33].

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 a 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 dependant 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) The must not be more than one process creation event with a given instance name.

□

Now, we specify conditions for a High Level Message Sequence Chart 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 ∧
/* 5 */
(∀ bmsc : BMSC′ • bmsc ∈ **rng** b ⇒ wf_BMSC(bmsc))
end

Annotations

- A HMSC is well-formed, if each of the following conditions hold.
- (1) The set of arrows must emanate from BMSCs that are in the mapping of BMSC names to BMSCs.
- (2) The set of arrows must terminate at BMSCs that are in the mapping of BMSC names to BMSCs.
- (3) The start node must be in the mapping of BMSC names to BMSCs.
- (4) The end nodes must be in the mapping of BMSC names to BMSCs.
- (5) All BMSCs in the HMSC must be well-formed.

□

4 Statecharts

In this section we describe Statecharts [8, 9] which is a graphical notation tailored for specifying the control part of reactive systems, i.e. event-driven systems which react to external stimuli. Statecharts extend conventional state machines and state diagrams with hierarchical states and ways of specifying concurrency and communication.

4.1 Description

Like Petri Nets, Statecharts are centered around *states* and *transitions*. A state represents a possible configuration of a system. The behaviour of the system to external stimuli depends on the state(s) it is currently in. States may be decomposed into substates using either AND or XOR decomposition. AND decomposition captures the property that when a system is in a given state, it must also be in all substates of that state. Conversely, XOR decomposition captures the property that when a system is in a given state, it must be in exactly one of the substates of that state.

A transition connects a state to another state. A transition is triggered by some *event* provided some *condition* is satisfied. When a transition occurs, control is transferred from the origin state to the destination state. If the origin of a transition is a state with substates, control is relinquished by all substates. If the destination of a transition is a state that is AND decomposed into substates, control is assumed by all substates. If the destination is XOR decomposed, control is assumed by the default substate.

A special kind of transition causes control to be transferred to the substate(s) that most recently had control instead of the default substate. This history-dependant type of transition may be related only to the immediate substates, or recursively to substates, substates of substates, etc. In cases where no history is available, i.e. the first time control is transferred to a state, the default substates are used.

The events that trigger transitions are typically the result of external stimuli, but may also be generated by timeouts when control has been in a given state for a predetermined period of time.

Statecharts are represented graphically as *higraphs* [9]. States are represented as rounded rectangles (called boxes in the following) and transitions are represented as arrows between boxes. Hierarchy is represented using area inclusion rather than the conventional tree or graph structures. A box whose outline is contained entirely within the outline of another box represents a substate of the containing state. XOR decomposition is represented by having several substates. AND decomposition is represented by subdividing the box of the containing state with a dashed line and placing concurrent substates on either side of the line. Default states are indicated by a small filled circle with an arrow pointing to the box of the default substate. History-dependent transitions are represented by letting the arrow of the transition point to a symbol composed of an 'H' inside a circle. If the transition depends on the recursive history, the H-symbol is decorated with an asterisk.

Example 6. Figure 8 shows an example statechart modelling a reactive system that receives four kinds of stimuli from the environment (essentially like four buttons). Each kind of stimuli generates a unique event, called a, b, c and d.

The system is represented by state A. Initially, the system is in state B. When an event d occurs, control is transferred to state F. A b event will now transfer control to state E. An additional b event will transfer control to state G and H, which is the default substate of G, while a c event will transfer control back to state F. When the system is in any of the substates of G, a b event will cause control to be transferred to F.

When the system is in any substate of D an a event will transfer control to both J and K. Similarly, when another b event occurs, control is relinquished by both J and K

and all their substates. Control is then transferred to the most resently visited states in D down to the lowest level, i.e. the states from which control was relinquished when the last a event occurred.

The label "/clh(C^*)" on the transition from B to C is an action that indicates that when this transition occurs, the history of C and its substates is deleted all the way down to the lowest level. Thus, the next time the transition from J/K to D occurs, the default state will be entered.

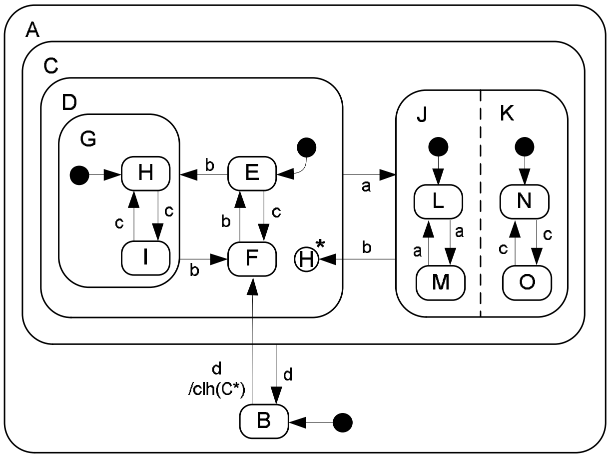

Figure 8: Example statechart.

□

4.2 Formalisation

The above description of statecharts is formalised in RSL.

scheme Statechart =
  class
    type
      Statechart = {| sc : Statechart′ • wf_Statechart(sc) |},
      Statechart′ = StateId × StateHier × Trans × History,
      StateHier = StateId $\underset{m}{\rightarrow}$ StateDef,
      StateDef == mk_XOR(OptSId, StateId-**set**) | mk_AND(StateId-**set**),
      OptSId == mk_None | mk_Id(StateId),
      Trans = StateId $\underset{m}{\rightarrow}$ Tr-**set**,
      Tr ::
        stid : StateId
        typ : Type
        evt : Event
        cond : Condition
        act : Action,
      Type == mk_History | mk_HistoryRec | mk_Direct,
      Event = **Text**,
      Condition,
      Action,
      History = StateId $\underset{m}{\rightarrow}$ StateId,
      StateId

```
value
  wf_Statechart : Statechart′ → Bool
  wf_Statechart(sid, shi, tr, hi) ≡
    /∗ 1 ∗/
    sid ∈ dom shi ∧
    /∗ 2 ∗/
    dom tr ⊆ dom shi ∧
    /∗ 3 ∗/
    dom hi ⊆ dom shi ∧
    /∗ 4 ∗/
    (∀ s : StateId •
       s ∈ dom shi ⇒
         case shi(s) of
           mk_XOR(os, ss) → ss ⊆ dom shi,
           mk_AND(ss) → ss ⊆ dom shi
         end) ∧
    /∗ 5 ∗/
    (∀ s : StateId •
       s ∈ dom shi ⇒
         case shi(s) of
           mk_XOR(os, ss) →
             case os of
               mk_None → true,
               mk_Id(sid′) → sid′ ∈ ss
             end,
           mk_AND(ss) → true
         end) ∧
    /∗ 6 ∗/
    (∀ s : StateId •
       s ∈ dom shi ⇒
         case shi(s) of
           mk_XOR(os, ss) →
             ss ≠ {} ⇒
               (∃ s′ : StateId, t : Tr •
                   s′ ∈ dom tr ∧ t ∈ tr(s′) ∧ stid(t) = s) ⇒
                 has_default(s, shi),
           mk_AND(ss) →
             ss ≠ {} ⇒
               (∃ s′ : StateId, t : Tr •
                   s′ ∈ dom tr ∧ t ∈ tr(s′) ∧ stid(t) = s) ⇒
                 has_default(s, shi)
         end),

  has_default : StateId × StateHier → Bool
  has_default(sid, shi) ≡
    case shi(sid) of
      mk_XOR(os, ss) →
        ss = {} ∨
        case os of
          mk_None → false,
```

$$
\begin{array}{l}
\qquad\qquad\qquad\quad \text{mk\_Id}(\text{sid}') \rightarrow \text{has\_default}(\text{sid}', \text{shi}) \\
\qquad\qquad\qquad \textbf{end}, \\
\qquad\qquad \text{mk\_AND}(\text{ss}) \rightarrow \\
\qquad\qquad\qquad \text{ss} = \{\} \vee (\forall\ \text{s} : \text{StateId} \bullet \text{s} \in \text{ss} \Rightarrow \text{has\_default}(\text{s}, \text{shi})) \\
\qquad\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

Annotations

- A statechart consists of an initial state identifier, a state hierarchy, a set of transitions and a history.
- A state hierarchy maps state identifiers to state definitions.
- A state definition is either an exclusive-or state or a both-and state. An exclusive-or state has an optional default substate identifier and a set of identifiers of substates. A both-and state has a set of identifiers of substates. A substate is a state.
- From a state identifier the set of transitions emanating from that state can be found.
- A transition has a destination stated identifier, a type, a triggering event, a condition and an action.
- A transition may either cause a transfer of control to the most recently visited state at the top-level, or a recursive transfer of control to the most recently visited state all the way down to the lowest level, or a direct transfer of control to the destination state.
- An event is a textual label identifying an exterior interaction.
- Conditions, actions and state identifiers are further undefined entities.
- The history is a mapping from state identifiers to substate identifiers.
- A statechart is wellformed, if
- (1) the initial state is in the state hierarchy, and;
- (2) the states from which transitions emanate are in the state hierarchy, and;
- (3) the states with a history are in the state hierarchy, and;
- (4) all substates are in the state hierarchy, and;
- (5) when an exclusive-or state has a default substate, then that substate is in the state hierarchy, and;
- (6) if a transition terminates at a composite state, then that state has a default substate.
- A state has a default state, if it has no substates, or if it is an exclusive-or state and it has a default substate which in turn has a default state, or if it is a both-and state and all its substates have a default state.

$\square$

5 Examples

In this section we present a number of examples of modelling using the graphical notations discussed in the previous sections. For each model we develop the graphical model and a similar RSL model in parallel.

5.1 Coloured Petri Nets – Superscalar Processor

We choose to illustrate Coloured Petri Nets by modelling a MIPS R10000 superscalar microprocessor [34]. The model presented is a simplification of the real processor, but includes enough detail to give a good picture of the control part of the processor. The CPN model is based on a model by Yakovlev *et al* [1].

5.1.1 Description

The R10000 is a 64-bit RISC microprocessor implementing the MIPS 4 instruction set. The processor fetches and decodes 4 instructions per cycle and employs branch prediction. The processor has five fully pipelined execution units: one load/store unit, two 64-bit integer ALUs, one 64-bit IEEE 754-1985 floating point adder and one 64-bit floating point multiplier. The entire pipeline has 7 stages. The R10000 has 33 64-bit logical integer registers and 32 64-bit logical floating point registers.

Instructions are issued and executed out of order which may lead to data hazards. Data hazards arise when two instructions reference the same register. If these instructions are executed out of order, the meaning of the program may be altered. To avoid data hazards the processor analyses dependencies among instructions and stalls instructions when there is a read-after-write (RAW), write-after-read (WAR) or write-after-write (WAW) dependency.

When a conditional jump instruction is encountered, the processor attempts to predict the direction of the jump before the condition is evaluated. The prediction algorithm simply assumes the branch will go in the same direction as it did the last time it was evaluated. If it has has not been evaluated before, it is assumed that no branch occurs. Subsequent instructions are fetched from the predicted direction and executed speculatively. When the value of the condition is later evaluated, it is checked whether the branch prediction was correct. If the prediction was incorrect, instructions following the jump are cleared from the pipeline and new instructions are fetched from the other direction of the jump. Data evaluated by instructions following a predicted branch is only written back to registers when the prediction has been confirmed. The R10000 may execute speculatively with up to 4 unconfirmed branches at a time.

The MIPS instruction set uses three operand instructions, i.e. all arithmetic instructions take three arguments: two source registers and one destination register.

The integer ALUs have a dual stage pipeline, so they can operate on two instructions at a time. The floating point units have a four stage pipeline. The address unit has a three stage pipeline.

5.1.2 Coloured Petri Net Model

Figure 9 shows the CPN model of the R10000 microprocessor. The accompanying colour set declarations and function definitions are given below.

The model is divided into five phases: instruction fetch, decoding, issue, execution and writeback.

In the fetch phase, instructions are loaded from memory (represented by the place *In*) one at a time. With each instruction loaded the program counter (*PC1*) is incremented by one. Instructions are buffered in the instruction queue (*Instr. Queue*) which may hold up to four instructions (this limit is enforced using the place *Queue Limit*).

In the decode phase, the instructions are labelled with a cache generation number. The cache generation number is incremented each time a conditional jump instruction is decoded and is used to ensure that instructions which depend on unconfirmed branches are not written back before the branches are confirmed. The program counter *PC2* is used to ensure instructions are decoded in-order. The place *Bra Pred* holds a branch prediction table, which records the direction of each of the previously evaluated branches. If a branch instruction is decoded, the program counters are updated to the address the branch is predicted to go to.

In the issue phase, instructions are issued to one of the five execution units. Instructions are only issued if they have no dependencies with any of the currently executing instructions and the required execution unit is available. Instructions may be issued out of order when some instructions are stalled.

In the execution phase, instructions are evaluated simultaneously in the five execution units and the results are stored temporarily.

In the writeback phase, instruction results are stalled until all predicted branches on which they depend have been confirmed. This is done by letting the place *CG2* record the highest cache generation where all branches have been resolved. When jump instructions pass the writeback phase, the branch prediction table is updated to reflect the direction taken by the jump and the cache generation is incremented. The place *Eval* is used to simulate the evaluation of conditions by randomly producing either the value true or false.

```
color
  Line         = int;
  PC           = int;
  Cachegen     = int;
  Value        = bool;
  InstType     = with INT | ADD | MULT | LS | BRA;
  Inst         = record
                   no : Line *
                   instr : InstType *
                   sou1 : Reg *
                   sou2 : Reg *
                   tar : RegLine *
                   cachegen : Cachegen;
  IReg         = index ireg with 1..33;
  FReg         = index freg with 1..32;
  Reg          = union ir : IReg + fr : FReg;
  RegLine      = union reg : Reg + line : Line;
  BraPredElem  = record
                   no : Line *
                   jmp : Line;
  BraPred      = list BraPredElem;
  FUnit        = with ALU1 | ALU2 | FPADD | FPMULT | ADR;
  Limit        = with token;
```

```
(* updpc : Inst * BraPred * PC → PC *)
updpc(inst, bp, pc) = if #instr inst=BRA then
                          let val a =lookup(#no inst, bp) in
                              if a<>0 then a else pc+1
                          end
                      else pc

(* updpc2 : Inst * BraPred * PC → PC *)
updpc2(inst, bp, pc2) = if #instr inst=BRA then
                            let val a =lookup(#no inst, bp) in
```

```
                    if a<>0 then a else pc2+1
                end
            else pc2+1

(* lookup : Line * BraPred → Line *)
lookup(no, bp) = if bp=[] then 0 else
                 if #no (hd bp) =no then #jmp (hd bp) else lookup(no, tl bp)

(* addtag : Inst * BraStack → Inst *)
addtag(inst, cg) = { no=#no inst,
                     instr=#instr inst,
                     sou1=#sou1 inst,
                     sou2=#sou2 inst,
                     tar=#tar inst,
                     cachegen=cg }

(* regdeps : Inst → Reg *)
regdeps(inst) = case #instr inst of
                  BRA    ⇒ empty
                | LS     ⇒ #tar inst
                | MULT   ⇒ 1'(#sou1 inst) +1'(#sou2 inst) +1'(#tar inst)
                | ADD    ⇒ 1'(#sou1 inst) +1'(#sou2 inst) +1'(#tar inst)
                | INT    ⇒ 1'(#sou1 inst) +1'(#sou2 inst) +1'(#tar inst)

(* issfu : Inst * FUnit * FUnit → Inst *)
issfu(inst, fu, fud) = if fu=fud then inst else empty

(* eval : unit → bool *)
eval() = Random.rnd() <0.5

(* evaljmp : Inst * bool → Line *)
evaljmp(inst, tr) = if tr then #line (#tar inst) else (#no inst)+1

(* upbp : Inst * Line * BraPred → BraPred *)
upbp(inst, a, bp) = if #instr inst=BRA then
                      if bp=[] then [{no=#no inst, jmp=a}] else
                        if #no (hd bp)=#no inst then [{no=#no inst, jmp=a}] :: (tl bp)
                        else (hd bp) :: updp(inst, a, tl bp)
                    else bp

(* updcg : Inst * Cachegen → Cachegen *)
updcg1(inst, cg) = if #instr inst=BRA then cg+1 else cg

(* updcg2 : Inst * Cachegen → Cachegen *)
updcg2(inst, cg) = if #instr inst=BRA then #cachegen wb else cg

(* ig : InstType * FUnit → bool *)
ig(instt, fu) = case instt of
                  BRA    ⇒ ic=ALU1 orelse ic=ALU2
                | LS     ⇒ ic=ADR
                | MULT   ⇒ ic=FPMULT
                | ADD    ⇒ ic=FPADD
                | INT    ⇒ ic=ALU1 orelse ic=ALU2

(* wbg : Inst * Cachegen → bool *)
wbg(inst, cg) = (#instr inst=BRA andalso #cachegen inst≤cg+1) orelse #cachegen inst≤cg
```

5.1.3 RSL Model

We now present an RSL model corresponding to the CPN model. First, types corresponding to the colour sets of the CPN model are defined. Some of the types differ from their colour set counterpart and some colour sets are omitted entirely.

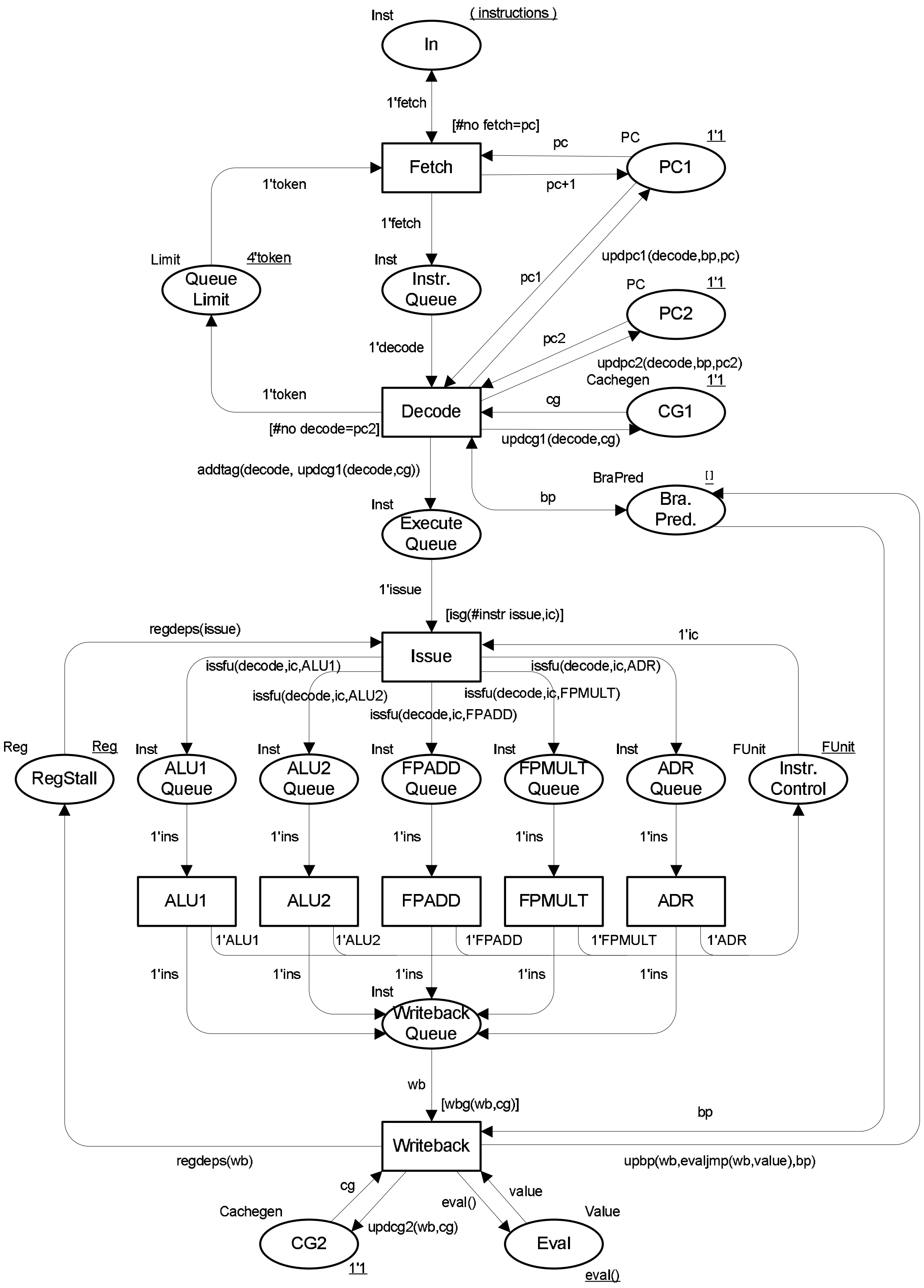

Figure 9: Simplified CPN model of the superscalar microprocessor MIPS R10000.

```
scheme SuperscalarProcessorTypes =
  class
    type
      Line = Int,
      PC = Int,
      Cachegen = Int,
      InstType == INT | ADD | MULT | LS | BRA,
      Inst ==
        mk_Inst(
          no : Line,
          instr : InstType,
          sou1 : Reg,
          sou2 : Reg,
          tar : RegLine,
          cachegen : Cachegen ↔ recon_cg),
      Reg =
        {| r : Reg′ •
          case r of
            IReg(n) → n ∈ {1 .. 33},
            FReg(n) → n ∈ {1 .. 32}
          end |},
      Reg′ == IReg(Nat) | FReg(Nat),
      RegLine = Reg | Line,
      BraPred = Line →m Line,
      FUnit == ALU1 | ALU2 | FPADD | FPMULT | ADR,
      Limit = Nat,
      LimitRel == Release,
      BraPredUpd == Update(Line, Line) ,
      InstReadyIs == Some(Nat, Reg-set, FUnit) | None,
      InstReadyWb == Some(Nat) | None
  end
```

Annotations

- Program line numbers, program counters and cache generations are integers.
- There are five types of instructions: integer instructions, floating point addition, floating point multiplication, load/store and conditional branch.
- An instruction is characterised by a line number, an instruction type, two source registers and a destination register, and a cache generation.
- A register is one of 33 integer registers or 32 floating point registers.
- The branch predition table maps branch instruction line numbers to the line they most recently caused a jump to.
- There are five functional units: two ALUs, one floating point adder, one floating point multiplier and an address unit.
- *Limit* is used to limit the number of instructions in the instruction queue to four.
- The signal *LimitRel* indicates that an instruction was removed from the instruction queue.

- The signal *BraPredUpd* indicates that the branch prediction table should be updated.
- *InstReadyIs* and *InstReadyWb* are returned by function indicating whether instructions are ready is the issue and writeback phases, respectively.

□

In moving from the CPN model to the RSL model we use the following principle: transitions become processes and places become parameters for processes. The principle is motivated by the observation that places are really buffers for values needed for computations, i.e. transitions. We aim to join places and transitions to form processes, such that the minimum amount of communication between processes is necessary.

Figure 10 illustrates the processes and channels in the RSL model. All channels are used for one-way communication only.

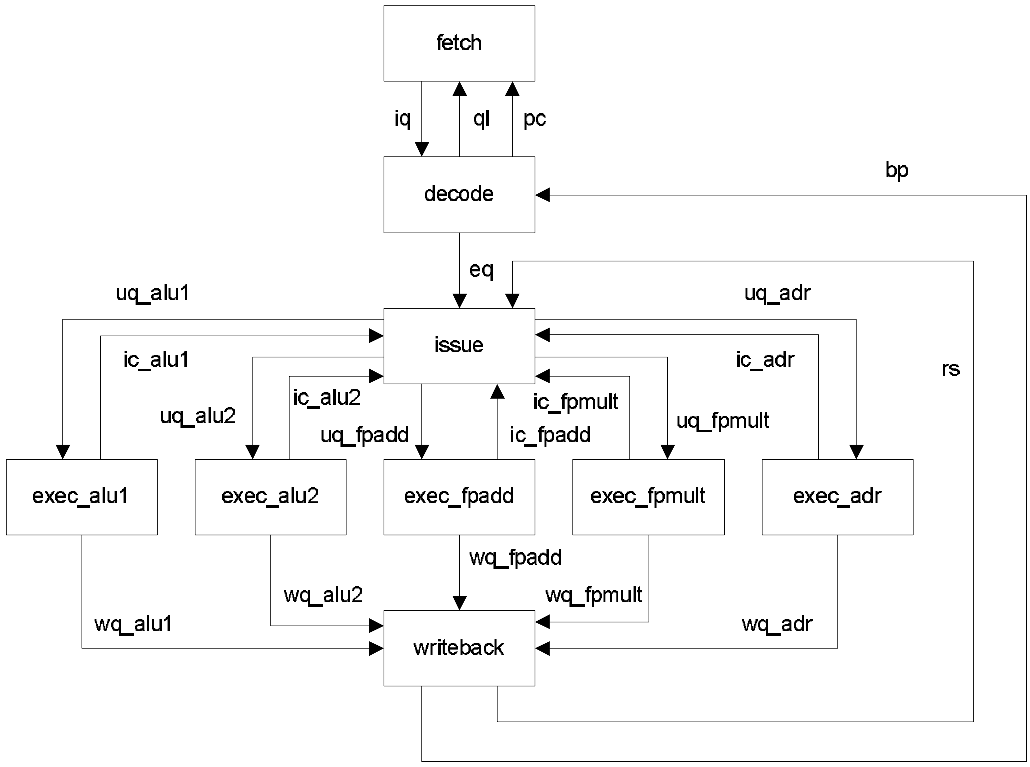

Figure 10: RSL channels and processes.

context: SuperscalarProcessorTypes

scheme SuperscalarProcessor =
  extend SuperscalarProcessorTypes **with**
  class
    channel
      iq : Inst,
      ql : LimitRel,
      pc : Line,
      eq : Inst,
      uq_alu1 : Inst,
      uq_alu2 : Inst,
      uq_fpadd : Inst,
      uq_fpmult : Inst,

```
    uq_adr : Inst,
    wq_alu1 : Inst,
    wq_alu2 : Inst,
    wq_fpadd : Inst,
    wq_fpmult : Inst,
    wq_adr : Inst,
    rs : Reg-set,
    ic_alu1 : FUnit,
    ic_alu2 : FUnit,
    ic_fpadd : FUnit,
    ic_fpmult : FUnit,
    ic_adr : FUnit,
    bp : BraPredUpd

value
    system : (Line -m-> Inst) → in any out any Unit
    system(prg) ≡
        fetch(prg, 4, 1)
        ‖
        decode(⟨⟩, [ ], 1, 1)
        ‖
        issue(
            ⟨⟩, {r | r : Reg},
            [ ALU1 ↦ 2, ALU2 ↦ 2, FPADD ↦ 4, FPMULT ↦ 4, ADR ↦ 3 ])
        ‖
        execute()
        ‖
        writeback(⟨⟩, 1),

    fetch : (Line -m-> Inst) × Limit × PC → in ql, pc out iq Unit
    fetch(prg, l, pc) ≡
        if pc ∉ dom prg then chaos
        else
            (if l > 0 then iq!prg(pc) ; fetch(prg, l − 1, pc + 1) end)
            ⌷
            (let q = ql? in fetch(prg, l + 1, pc) end)
            ⌷
            (let pc′ = pc? in fetch(prg, l, pc′) end)
        end,

    decode :
        Inst* × BraPred × PC × Cachegen → in iq, bp out ql, pc, eq Unit
    decode(il, bra, prc, cg) ≡
        if no(hd il) ≠ prc then decode(tl il, bra, prc, cg)
        else
            (let i = iq? in decode(il ⌢ ⟨i⟩, bra, prc, cg) end)
            ⌷
            (let Update(line, tar) = bp? in
                decode(il, bra † [ line ↦ tar ], prc, cg)
             end)
            ⌷
```

```
    (let i = hd il in
        if instr(i) = BRA
        then
            eq!recon_cg(cg + 1, i) ;
            ql!Release ;
            if prc ∈ dom bra
            then pc!bra(prc) ; decode(tl il, bra, bra(prc), cg + 1)
            else decode(tl il, bra, prc + 1, cg + 1)
            end
        else
            eq!recon_cg(cg, i) ; ql!Release ; decode(tl il, bra, prc + 1, cg)
        end
    end)
  end,

issue :
  Inst* × Reg-set × (FUnit →m Nat) →
    in eq, rs, ic_alu1, ic_alu2, ic_fpadd, ic_fpmult, ic_adr
    out uq_alu1, uq_alu2, uq_fpadd, uq_fpmult, uq_adr Unit
issue(il, regs, units) ≡
  (let i = eq? in issue(il ⌢ ⟨i⟩, regs, units) end)
  []
  (let r = rs? in issue(il, regs ∪ r, units) end)
  []
  (let
      u = ic_alu1? [] ic_alu2? [] ic_fpadd? [] ic_fpmult? [] ic_adr?
   in
      issue(il, regs, units † [u ↦ units(u) + 1])
   end)
  []
  (case findready_is(1, il, regs, units) of
      None → issue(il, regs, units),
      Some(n, re, un) →
        (case un of
             ALU1 → uq_alu1!il(n),
             ALU2 → uq_alu2!il(n),
             FPADD → uq_fpadd!il(n),
             FPMULT → uq_fpmult!il(n),
             ADR → uq_adr!il(n)
          end) ;
        issue(remove(n, il), regs \ re, units † [un ↦ units(un) − 1])
   end),

execute : Unit → in any out any Unit
execute() ≡
  exec_alu1(⟨⟩)
  ||
  exec_alu2(⟨⟩)
  ||
  exec_fpadd(⟨⟩)
  ||
```

```
    exec_fpmult(⟨⟩)
    ‖
    exec_adr(⟨⟩),

exec_alu1 : Inst* → in uq_alu1 out wq_alu1, ic_alu1 Unit
exec_alu1(il) ≡
    (let i = uq_alu1? in exec_alu1(il ⌢ ⟨i⟩) end)
    ⌷
    (wq_alu1!hd il ; ic_alu1!ALU1 ; exec_alu1(tl il)),

exec_alu2 : Inst* → in uq_alu2 out wq_alu2, ic_alu2 Unit
exec_alu2(il) ≡
    (let i = uq_alu2? in exec_alu2(il ⌢ ⟨i⟩) end)
    ⌷
    (wq_alu2!hd il ; ic_alu2!ALU2 ; exec_alu2(tl il)),

exec_fpadd : Inst* → in uq_fpadd out wq_fpadd, ic_fpadd Unit
exec_fpadd(il) ≡
    (let i = uq_fpadd? in exec_fpadd(il ⌢ ⟨i⟩) end)
    ⌷
    (wq_fpadd!hd il ; ic_fpadd!FPADD ; exec_fpadd(tl il)),

exec_fpmult : Inst* → in uq_fpmult out wq_fpmult, ic_fpmult Unit
exec_fpmult(il) ≡
    (let i = uq_fpmult? in exec_fpmult(il ⌢ ⟨i⟩) end)
    ⌷
    (wq_fpmult!hd il ; ic_fpmult!FPMULT ; exec_fpmult(tl il)),

exec_adr : Inst* → in uq_adr out wq_adr, ic_adr Unit
exec_adr(il) ≡
    (let i = uq_adr? in exec_adr(il ⌢ ⟨i⟩) end)
    ⌷
    (wq_adr!hd il ; ic_adr!FPMULT ; exec_adr(tl il)),

writeback :
    Inst* × Cachegen →
        in wq_alu1, wq_alu2, wq_fpadd, wq_fpmult, wq_adr out bp Unit
writeback(il, cg) ≡
    (let
        i = wq_alu1? ⌷ wq_alu2? ⌷ wq_fpadd? ⌷ wq_fpmult? ⌷ wq_adr?
     in
        writeback(il ⌢ ⟨i⟩, cg)
     end)
    ⌷
    (case findready_wb(1, il, cg) of
        None → writeback(il, cg),
        Some(n) →
            let i = il(n), valu = random() in
                if instr(i) = BRA
                then
                    bp!Update(n, evaljmp(il(n), valu)) ;
```

```
              writeback(remove(n, il), cg + 1)
            else writeback(remove(n, il), cg)
            end
          end
      end),

findready_is :
   Int × Inst* × Reg-set × (FUnit →m Nat) → InstReadyIs
findready_is(j, il, rs, fu) ≡
   if il = ⟨⟩ then None
   else
      let i = hd il in
         case instr(i) of
            INT →
               if {sou1(i), sou2(i), RegLine_to_Reg(tar(i))} ⊆ rs
               then
                  if fu(ALU1) > 0
                  then Some(j, {sou1(i), sou2(i), RegLine_to_Reg(tar(i))}, ALU1)
                  else
                     if fu(ALU2) > 0
                     then
                        Some(j, {sou1(i), sou2(i), RegLine_to_Reg(tar(i))}, ALU1)
                     else findready_is(j + 1, tl il, rs, fu)
                     end
                  end
               else findready_is(j + 1, tl il, rs, fu)
               end,
            ADD →
               if {sou1(i), sou2(i), RegLine_to_Reg(tar(i))} ⊆ rs
               then
                  if fu(FPADD) > 0
                  then
                     Some(j, {sou1(i), sou2(i), RegLine_to_Reg(tar(i))}, FPADD)
                  else findready_is(j + 1, tl il, rs, fu)
                  end
               else findready_is(j + 1, tl il, rs, fu)
               end,
            MULT →
               if {sou1(i), sou2(i), RegLine_to_Reg(tar(i))} ⊆ rs
               then
                  if fu(FPMULT) > 0
                  then
                     Some(j, {sou1(i), sou2(i), RegLine_to_Reg(tar(i))}, FPMULT)
                  else findready_is(j + 1, tl il, rs, fu)
                  end
               else findready_is(j + 1, tl il, rs, fu)
               end,
            LS →
               if {RegLine_to_Reg(tar(i))} ⊆ rs
               then
                  if fu(ADR) > 0 then Some(j, {RegLine_to_Reg(tar(i))}, ADR)
```

```
                        else findready_is(j + 1, tl il, rs, fu)
                        end
                    else findready_is(j + 1, tl il, rs, fu)
                    end,
                BRA →
                    if fu(ALU1) > 0 then Some(j, {}, ALU1)
                    else
                        if fu(ALU2) > 0 then Some(j, {}, ALU2)
                        else findready_is(j + 1, tl il, rs, fu)
                        end
                    end
            end
        end
    end,

  findready_wb : Int × Inst* × Cachegen → InstReadyWb
  findready_wb(j, il, cg) ≡
    if il = ⟨⟩ then None
    else
        if
            cachegen(hd il) ≤ cg ∨
            (instr(hd il) = BRA ∧ cachegen(hd il) ≤ cg + 1)
        then Some(j)
        else findready_wb(j + 1, il, cg)
        end
    end,

  evaljmp : Inst × Bool → Line
  evaljmp(i, t) ≡ if t then RegLine_to_Line(tar(i)) else no(i) + 1 end,

  remove : Nat × Inst* → Inst*
  remove(n, il) ≡
    if il = ⟨⟩ then ⟨⟩
    else if n = 1 then tl il else ⟨hd il⟩ ⌢ remove(n − 1, tl il) end
    end,

  random : Unit → Bool
end
```

Annotations

- The process *system* starts all the remaining processes in parallel.
- The *fetch* process will either output an instruction to the *decode* process and update its program counter, or input a *LimitRel* signal and increase the number of free slots in the instruction queue by one, or input a new value of the program counter.
- The *decode* process will either input an instruction from *fetch*, or input a signal to update the branch prediction table, or output an instruction to the *issue* process updating the program counter if the instruction is a branch and branch prediction information is available.
- The *issue* process will either input an instruction from *decode*, or input a set of free registers from *writeback*, or input a free execution unit from one of the execution

unit processes, or issue an instruction to the proper execution unit, checking that no dependencies are violated.

- The *execute* process starts the five execution unit processes in parallel.
- The five execution unit processes are similar: they either input an instruction from *issue*, or output an instruction to *writeback* freeing one place in the execution unit.
- The *writeback* process either inputs an instruction from one of the execution unit processes, or finishes an instruction which does not depend on unconfirmed branch predictions.
- *findready_is* returns an instruction which may be issued without violating register dependencies or overloading the execution unit, or indicates that no such instruction exists.
- *findready_wb* returns an instruction which is ready for writeback, i.e. one which does not depend on unconfirmed branch predictions, or indicates that no such instruction exists.

□

5.2 Message Sequence Charts – 802.11 Wireless Network

We illustrate the use of Message Sequence Charts by modelling the possible exchanges of frames between an access point and a station in an IEEE 802.11 wireless network [22].

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 Figure 11 and the referenced BMSCs in Figure 12.

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. The access point decrypts the received encrypted nonce 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 its 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.

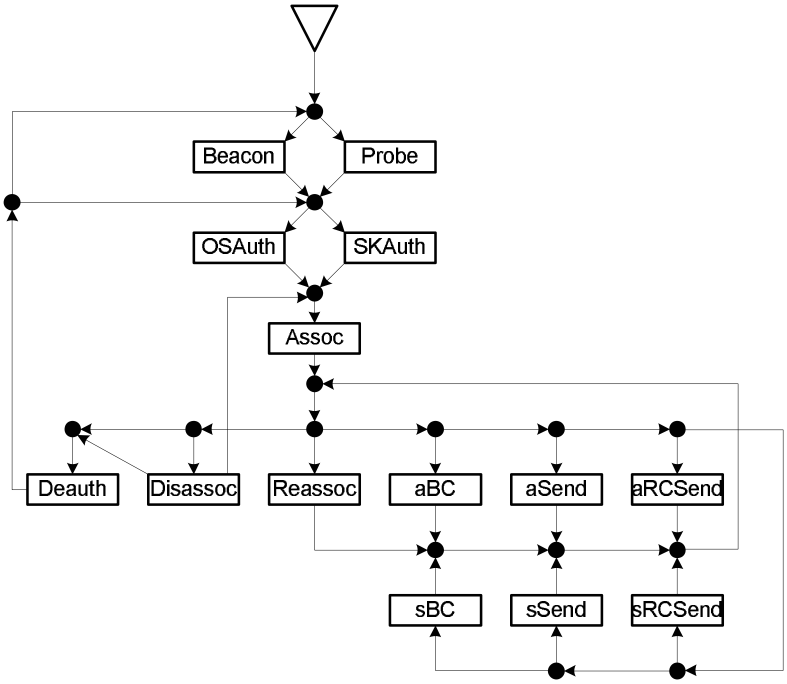

Figure 11: HMSC of 802.11 wireless network with an access point and a station.

5.2.1 RSL Model

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.

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 |
```

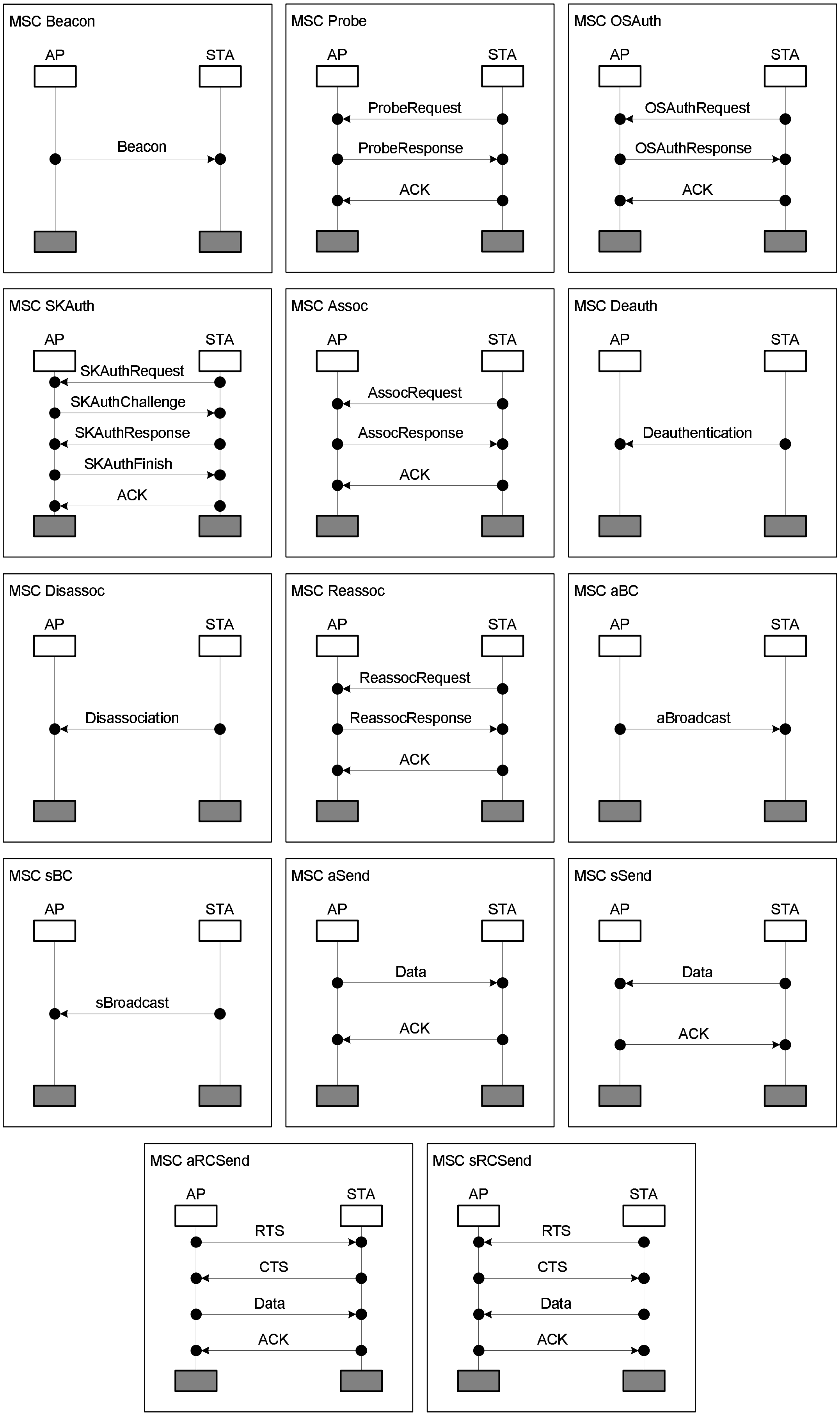

Figure 12: BMSCs referenced in Figure 11.

```
        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 it will participate in. 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 non-deterministic 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 the access point. Again, choices are abstracted as internal non-determinism.

```
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
```

5.3 Statecharts – Wireless Rain Gauge

In this section we present models of a wireless rain gauge.

5.3.1 Description

The rain gauge has two units: a container that collects and measures rain drops and a base station the display the measurements. The container is mounted outdoors, while the base station is placed indoors. The two units communicate via a radio signal. The base station is shown in Figure 13.

Figure 13: OBH wireless rain gauge base station.

The base station records the daily precipitation and keeps a history of the precipication for each of the previous 9 days. Also, it records the accumulated precipitation from a given start date. The base station also includes a digital clock.

The base station has three displays and four buttons that are used to set up the station and switch between its modes. Additionally the station has a *reset* button for restarting the station. The top display initially shows the current time and accumulated precipitation. If the button *mode/set* is pressed, the date is displayed, and if the button *since* is pressed, the start date of the accumulated precipitation measuring is displayed. The middle display indicates whether the base station receives a signal from the container. The button *search* is used to initiate a scan for a signal. The bottom display shows the daily precipitation and with repeated presses of button *history* the daily precipitation for each of the previous 9 days.

5.3.2 Statechart Model

Figure 14 shows the statechart for the rain gauge. In the initial state there are no batteries in the rain gauge. When batteries are inserted the three displays become operational. This is modelled by AND decomposition giving three concurrent states.

The date/time (top) display has four overall modes: time, date, start date of cumulative measuring and setup. Setup mode is entered by pressing and holding button *mode/set* for two seconds. If the button is released (indicated by the symbol $\hat{m}/s$ in the chart) before 2 seconds have elapsed, the date is displayed.

The signal (middle) display has three modes: either there is no signal, or a scanning is in progress, or there is a correct signal. A new scan may be initiated by pressing *search*.

The rain (bottom) display has two modes: normal operation showing precipitation for the current day, or history mode, where total precipitation for one of the last nine days is shown.

5.3.3 RSL Model

We translate the above statechart into RSL by creating a process for each state that has no substates. The process then responds to the events that cause transitions from

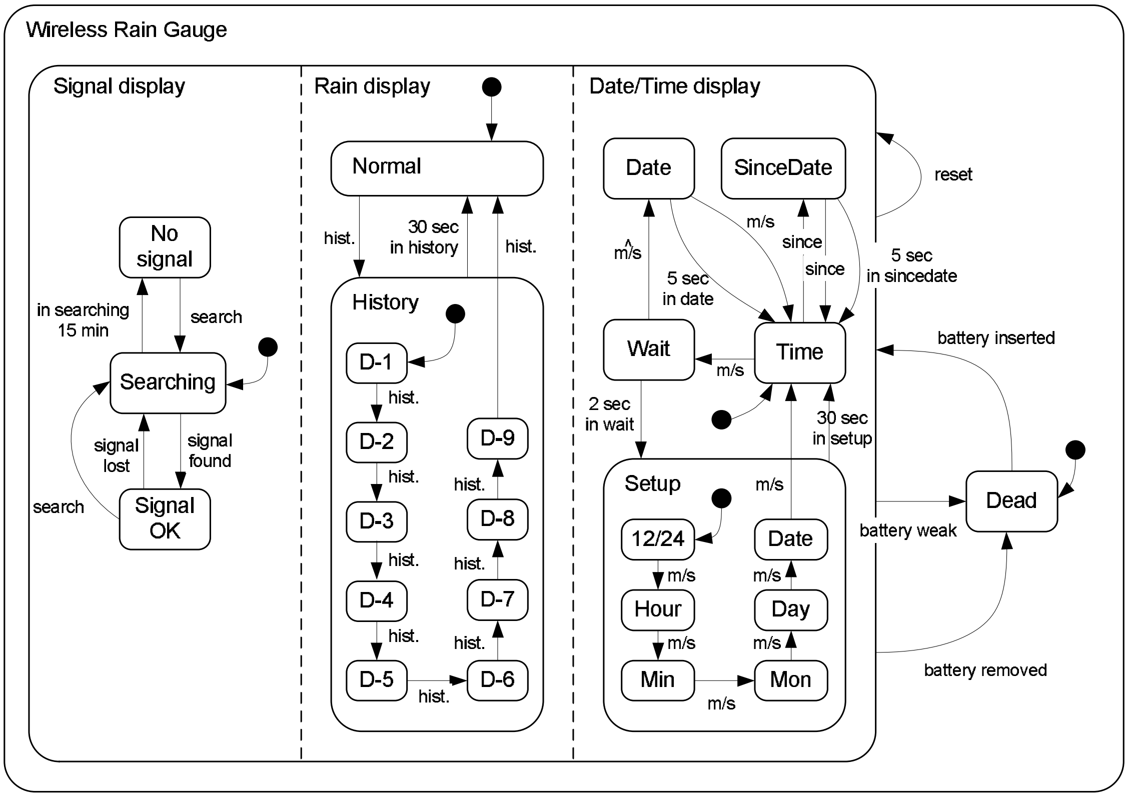

Figure 14: Statechart for wireless rain gauge.

the corresponding state. Special attention must be awarded to the AND composition. Whenever a transition causes several concurrent states to assume control, the translation in RSL starts multiple concurrent processes in parallel. Whenever several concurrent states lose control, the translation in RSL causes all but one of the concurrent processes to terminate. The single remaining process calls the process corresponding to the next state.

Timeouts are modelled as an external event. Note that the timeout durations specified in the statechart are lost in this translation to RSL. If this quantitative temporal information is to be preserved, the extension of RSL with Duration Calculus, called Timed RSL [35], may be used.

scheme RainGauge =
  class
    type
      Event ==
        BattIns |
        BattWeak |
        BattRem |
        SigFound |
        SigLost |
        ModeSet |
        ModeSetRel |
        History |
        Since |
        Search |
        Reset |
        Timeout

```
channel evt : Event

value
  Dead : Unit → in evt Unit
  Dead() ≡
    case evt? of
      BattIns → Searching() ∥ Normal() ∥ Time(),
      _ → Dead()
    end,

  SinceDate : Unit → in evt Unit
  SinceDate() ≡
    case evt? of
      Timeout → Time(),
      Since → Time(),
      Reset → Time(),
      BattRem → skip,
      BattWeak → skip,
      _ → SinceDate()
    end,

  Date : Unit → in evt Unit
  Date() ≡
    case evt? of
      Timeout → Time(),
      ModeSet → Time(),
      Reset → Time(),
      BattRem → skip,
      BattWeak → skip,
      _ → Date()
    end,

  Wait : Unit → in evt Unit
  Wait() ≡
    case evt? of
      Timeout → S_1224(),
      ModeSetRel → Date(),
      Reset → Time(),
      BattRem → skip,
      BattWeak → skip,
      _ → Wait()
    end,

  Time : Unit → in evt Unit
  Time() ≡
    case evt? of
      ModeSet → Wait(),
      Since → SinceDate(),
      Reset → Time(),
      BattRem → skip,
```

```
        BattWeak → skip,
        _ → Time()
    end,

S_1224 : Unit → in evt Unit
S_1224() ≡
    case evt? of
        Timeout → Time(),
        ModeSet → S_Hour(),
        Reset → Time(),
        BattRem → skip,
        BattWeak → skip,
        _ → S_1224()
    end,

S_Hour : Unit → in evt Unit
S_Hour() ≡
    case evt? of
        Timeout → Time(),
        ModeSet → S_Min(),
        Reset → Time(),
        BattRem → skip,
        BattWeak → skip,
        _ → S_Hour()
    end,

S_Min : Unit → in evt Unit
S_Min() ≡
    case evt? of
        Timeout → Time(),
        ModeSet → S_Mon(),
        Reset → Time(),
        BattRem → skip,
        BattWeak → skip,
        _ → S_Min()
    end,

S_Mon : Unit → in evt Unit
S_Mon() ≡
    case evt? of
        Timeout → Time(),
        ModeSet → S_Day(),
        Reset → Time(),
        BattRem → skip,
        BattWeak → skip,
        _ → S_Mon()
    end,

S_Day : Unit → in evt Unit
S_Day() ≡
    case evt? of
```

```
        Timeout → Time(),
        ModeSet → S_Date(),
        Reset → Time(),
        BattRem → skip,
        BattWeak → skip,
        _ → S_Day()
    end,

S_Date : Unit → in evt Unit
S_Date() ≡
    case evt? of
        Timeout → Time(),
        ModeSet → Time(),
        Reset → Time(),
        BattRem → skip,
        BattWeak → skip,
        _ → S_Date()
    end,

Normal : Unit → in evt Unit
Normal() ≡
    case evt? of
        History → H_D1(),
        Reset → Normal(),
        BattRem → Dead(),
        BattWeak → Dead(),
        _ → Normal()
    end,

H_D1 : Unit → in evt Unit
H_D1() ≡
    case evt? of
        Timeout → Normal(),
        History → H_D2(),
        Reset → Normal(),
        BattRem → Dead(),
        BattWeak → Dead(),
        _ → H_D1()
    end,

H_D2 : Unit → in evt Unit
H_D2() ≡
    case evt? of
        Timeout → Normal(),
        History → H_D3(),
        Reset → Normal(),
        BattRem → Dead(),
        BattWeak → Dead(),
        _ → H_D2()
    end,
```

```
H_D3 : Unit → in evt Unit
H_D3() ≡
   case evt? of
      Timeout → Normal(),
      History → H_D4(),
      Reset → Normal(),
      BattRem → Dead(),
      BattWeak → Dead(),
      _ → H_D3()
   end,

H_D4 : Unit → in evt Unit
H_D4() ≡
   case evt? of
      Timeout → Normal(),
      History → H_D5(),
      Reset → Normal(),
      BattRem → Dead(),
      BattWeak → Dead(),
      _ → H_D4()
   end,

H_D5 : Unit → in evt Unit
H_D5() ≡
   case evt? of
      Timeout → Normal(),
      History → H_D6(),
      Reset → Normal(),
      BattRem → Dead(),
      BattWeak → Dead(),
      _ → H_D5()
   end,

H_D6 : Unit → in evt Unit
H_D6() ≡
   case evt? of
      Timeout → Normal(),
      History → H_D7(),
      Reset → Normal(),
      BattRem → Dead(),
      BattWeak → Dead(),
      _ → H_D6()
   end,

H_D7 : Unit → in evt Unit
H_D7() ≡
   case evt? of
      Timeout → Normal(),
      History → H_D8(),
      Reset → Normal(),
      BattRem → Dead(),
```

```
        BattWeak → Dead(),
        _ → H_D7()
    end,

H_D8 : Unit → in evt Unit
H_D8() ≡
    case evt? of
        Timeout → Normal(),
        History → H_D9(),
        Reset → Normal(),
        BattRem → Dead(),
        BattWeak → Dead(),
        _ → H_D8()
    end,

H_D9 : Unit → in evt Unit
H_D9() ≡
    case evt? of
        Timeout → Normal(),
        History → Normal(),
        Reset → Normal(),
        BattRem → Dead(),
        BattWeak → Dead(),
        _ → H_D9()
    end,

NoSignal : Unit → in evt Unit
NoSignal() ≡
    case evt? of
        Search → Searching(),
        Reset → Searching(),
        BattRem → skip,
        BattWeak → skip,
        _ → NoSignal()
    end,

Searching : Unit → in evt Unit
Searching() ≡
    case evt? of
        Timeout → NoSignal(),
        SigFound → SignalOK(),
        Reset → Searching(),
        BattRem → skip,
        BattWeak → skip,
        _ → Searching()
    end,

SignalOK : Unit → in evt Unit
SignalOK() ≡
    case evt? of
        Search → Searching(),
```

```
            SigLost → Searching(),
            BattRem → skip,
            BattWeak → skip,
            _ → SignalOK()
         end
end
```

6 Conclusion

In this report we have described and formally defined the syntax and static semantics of the graphical specification techniques of Petri Nets, Message Sequence Charts and Statecharts. We have studied three larger examples where we have developed a model using one of the graphical notations and a corresponding model using the specification language RSL. We have specified the models such that they convey the same information, i.e. we have refrained from adding extra detail to one of the models even if one of the notations encourages such extensions.

Coloured Petri Nets is a very expressive and powerful modelling tool and like RSL, it has a formal mathematical semantics that allows properties of a specification to be proven formally. Another appealing feature of Coloured Petri Nets is that simulators and analysers are available. Such tools allow a specification to be tested directly.

Message Sequence Charts is a rather simple notation which in itself has limited expressibility. The most recently standardised version of Message Sequence Charts describes how to couple charts with syntax definitions in ASN.1. This is certainly an improvement, but a way of specifying operations and processes is still missing and must be added on an *ad hoc* basis. The advantage of Message Sequence Charts is that they are simple to grasp, thus allowing a client to validate and verify such charts.

Statecharts are well suited for specifying control flow in reactive systems, where state transitions are triggered by external events. Although we have no discussed it in this presentation, actions may be added to transitions so that a statechart can also specify data flow.

We have seen that all three graphical techniques can be expressed in terms of RSL[2], showing that RSL is at least as expressive as the union of the three graphical techniques. Combining the visual intuition of diagrams with the formalism and expressibility of RSL would provide a strong integrated specification technique with wide applications.

[2]Although, in order to model quantitative temporal aspects, we have to use Timed RSL instead of standard RSL

References

[1] F.P. Burns, A.M. Koelmans, and A.V. Yakovlev. Analysing superscalar processor architectures with coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):182–191, 1998.

[2] CCITT. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.

[3] Zhou Chaochen and Michael R. Hansen. *Duration Calculus: A formal approach to real–time systems*. Monographs in Theoretical Computer Science. Springer–Verlag, 2002 (2003). To be published.

[4] Meta Software Corporation. Design/CPN Reference Manual. Available at http://www.daimi.au.dk/designCPN/man/Reference/Reference.All.pdf, 1993.

[5] Jörg Desel and Ekkart Kindler. Proving correctness of distributed algorithms – a petri net approach.

[6] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, (1):115–138, 1971.

[7] D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1985.

[8] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[9] David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.

[10] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.

[11] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[12] Anne Haxthausen and Xia Yong. Linking DC together with TRSL. In *Proceedings of 2nd International Conference on Integrated Formal Methods (IFM'2000), Schloss Dagstuhl, Germany, November 2000*, number 1945 in Lecture Notes in Computer Science, pages 25–44. Springer-Verlag, 2000.

[13] Anne Elisabeth Haxthausen. Some approaches for integration of specification techniques (invited extended abstract).

[14] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[15] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.

[16] Kurt Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use, Volume 1 Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[17] Ekkart Kindler, Wolfgang Reisig, Hagen Volzer, and Rolf Walter. Petri net based verification of distributed algorithms: An example. *Formal Aspects of Computing*, 9(4):409–424, 1997.

[18] Lars M. Kristensen, Soren Christensen, and Kurt Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[19] P. Ladkin and S. Leue. An analysis of message sequence charts, 1992.

[20] Peter B. Ladkin and Stefan Leue. What do message sequence charts mean? In *FORTE*, pages 301–316, 1993.

[21] Peter B. Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[22] IEEE LAN and MAN Standards Committee. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE, 1999. IEEE 802.11 Standard.

[23] L. Li and H. Jifeng. A denotational semantics of timed rsl using duration calculus, 1999.

[24] L. Li and H. Jifeng. Towards a denotational semantics of Timed RSL using duration calculus, 1999.

[25] Zohar Manna and Amir Pnueli. Models for reactivity. *Acta Informatica*, 30(7):609–678, 1993.

[26] S. Mauw and M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994.

[27] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[28] M. Naedele and J. Janneck. Design patterns in petri net system modeling, 1998.

[29] Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[30] W. Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, 1998.

[31] Wolfgang Reisig. *A Primer in Petri Net Design*. Springer-Verlag, 1992.

[32] M. Reniers. Static semantics of message sequence charts, 1995.

[33] M.A. Reniers. Syntax requirements of message sequence charts. In R. Braek and A. Sarma, editors, *Proceedings of the 7th SDL Forum*, 1995.

[34] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 40(2):28–40, 1996.

[35] Xia Yong and Chris W. George. An Operational Semantics for Timed RAISE. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 1008–1027. FME, Springer–Verlag, 1999.