Embedding ASMs into State Transition Diagrams

Theo Sattler, Wolfgang Ahrendt

July 7, 2000

Abstract

This report relates *Abstract State Machines* (ASMs) with a particular diagram type of UML, the *Sate Transition Diagrams* (STDs). The principles of translating ASMs into STDs are discussed and demonstrated in four case studies.

1 Introduction

In this report, two formalisms to model state based systems are related.

The first is the frame of *Abstract State Machines* (ASMs). ASMs, which formerly were called *Evolving Algebras*, are defined in [Gur95, Gur97]. We assume some familiarity with ASMs. For a general introduction, see [Bör99]. ASMs are used in many case studies to formalize, for instance, the behavior of systems, the semantics of languages, or the specification of hardware.

The second formalism, called *Sate Transition Diagrams* (STDs), forms one of the diagram types of the *Unified Modeling Language* (UML), which is becoming a de facto standard for the modeling of object oriented systems. STDs are defined in the UML specification [Obj99]. "Sate Transition Diagrams" basically is a new name for a formalism called "State Charts" (which did exist before UML), even if the object orientedness of UML caused some slight modifications. For syntax and semantics of Sate Charts, see [Har87, HN96].

Having worked with ASMs as well as with UML, we felt that it is natural to embed the first into the second, in form of particular STDs. At least, many examples of ASMs seem to be appropriate to be taken as an STD. We demonstrate this in four case studies, the first two based on the famous 'WAM paper' of Börger/Rosenzweig [BR94], the last two based on the Kermit protocol formalization of Huggins [Hug95].

We investigate the principles of translating ASMs, which are represented textually, into the graphical notation of STDs, thereby embedding ASMs into UML.

2 Background

Below, we briefly discuss the statics and dynamics of both, Abstract State Machines and State Transition Diagrams, emphasizing the differences between them.

Statics: the states

The concept of states is essentially different in ASMs and STDs. In an ASM, which essentially is a set of rules performing updates on functions, a state is a particular interpretation of all functions. Such a state gives a total characterization of the system at a certain time. States are not syntactically present in the rules.

In a Sate Transition Diagram, states are given as syntactical entities, being the nodes in a (hierarchical) graph. The states are meant to represent some general characteristics of the system. Attributes of the system are only partially characterized by the states. STDs can make use of orthogonal decomposition (see Figure 1). It is assumed that the system is in exactly one state of each of the orthogonal component diagrams. The compound states then are implicitly given by the product of the states of its components.

Dynamics: rules and transitions

Also, the concept of transitions and rules differs between ASMs, and STDs. ASM rules have the form "**if** *guard* **then** *act1, act2, ...* **else** *act3, act4, ...*". '**if**' statements can also be nested. Rules are not associated with a specific state, but apply globally in any state. To determine which rules can be fired in the current state, the guards are examined. Then, all rules whose guards are valid fire simultaneously. If updates in simultaneously fired rules contain contradicting updates, the ASM is called *inconsistent*.

STD transition have explicit source, and destination state. Transitions are pictured through an arrow from the source to the destination state. Guards, and actions are put on these arrow as "[*guard*] / *act1, act2, ...*" (see Figure 2). If the source, and the destination state are identical, then the transition is called internal. Such transitions are put inside the corresponding state box. An additional feature of STDs are events. The occurrence of events can be used as guard, and triggered in an action. Sending an event is pictured with a '^' sign (^ *receiver.event*).

At each step, only one of the transitions leaving the same state is allowed to fire. If two such transitions potentially could fire in the same step, they are said to be *in conflict*. Nonetheless, transitions in orthogonal components fire simultaneously. Also, multiple internal transitions can fire simultaneously. As in ASMs, the firing of multiple transitions can lead to contradicting updates.

Modules

Distributed ASMs provide the specification of distributed systems via modules, each modeling an independent ASM. Communication is established by public functions in each module which are modifiable by other modules. The updates in different modules are fired asynchronously, but still, the updates in a single module are applied simultaneously.

The module concept fits nicely into the object concept UML is based on. So, each module can be mapped into a corresponding class. The rules of each module are translated into a STD associated with these class.

Non-distributed ASMs can be seen as DASMs consisting of just one module.

Initialization

ASMs are assumed to run from a somehow defined initial state. The description of this state is an integral part of the ASM's syntax.

In STDs, the initial state is pictured with a bold dot (●). This state must be left immediately. Initial states are also used to picture the first state in component diagrams. Using orthogonal decomposition, each component has a individual initial state. Further initialization must be made explicitly via transitions. To model ASM initialization in STDs, it is advisable to use an additional dedicated initialization state which initializes the whole system, see below.

3 Translation of ASMs into STDs

3.1 States

To characterize the states of the STD, a subset of the ASM's functions is chosen. This functions (not necessarily 0-ary) must have a small (finite) image. Subsequently, this

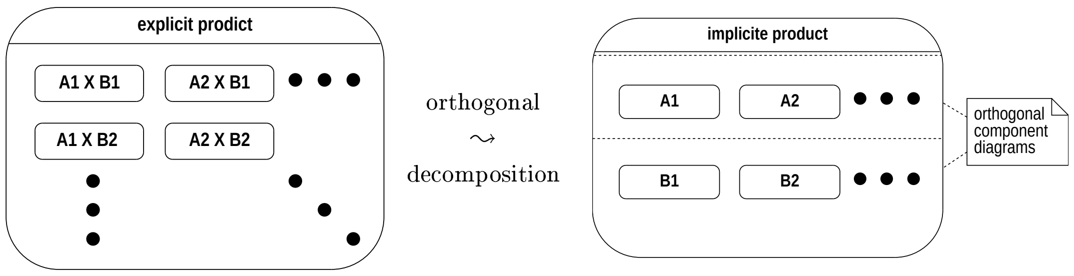


Figure 1: Orthogonal decomposition

functions will be referred to as *characterizing functions*. The set of possible states for the STD then is given by the product of these images.

Using the explicit product to picture the states in STDs rapidly leads to confusing diagrams. To simplify the diagram, it is advisable to utilize orthogonal decomposition, which is nicely provided by the STD notation. A suggestive splitting is given by generating one component diagram for each characterizing function. In Figure 1, two characterizing functions are assumed with image {A1,A2,...}, and {B1,B2,...} respectively (Notice, in this case the bold dots in the diagrams are not to read as initial states, but as periods). Subsequently, orthogonal decomposition is used by default.

3.2 Transitions

The translation of ASM rules into STD transitions is a bit more delicate. To ensure fitting functionality, several measures have to be taken. Below, the process of translation is split into several consecutive steps.

i) Expansion of ASM rules

ASMs provide nested conditions. Such conditions cannot be translated directly into STDs. Therefore, the nested **if-then-else** rules must be broken down into flat **if-then** rules. The resulting ASM rules now can be translated into STD transitions.

ii) Source, and destination states

ASM rules apply globally to all states. Therefore, primarily each ASM rule results in transitions for each state of the STD. This set of transitions can be restricted taking into account the characterizing functions appearing in the guard.

The destination state is determined through updates of characterizing functions. If no characterizing functions are updated, the destination state is identical with the source state.

iii) Classification of transitions

To classify transitions, their profile, i.e. the set of source and destination states is examined. Source and destination states are determined by characterizing functions in the guard and the updates, respectively. Transitions are classified depending on the profiles.

To exemplify the following classification, a simple ASM is used. It has two characterizing functions *ch_a*, and *ch_b* with images {*A1,A2,A3*}, and {*B1,B2*}, respectively. The corresponding STD is given in Figure 2. For each class, example rules together with their profile are provided. The examples only contain characterizing functions. Guards, and actions not depending on characterizing functions do neither influence the profile, nor the

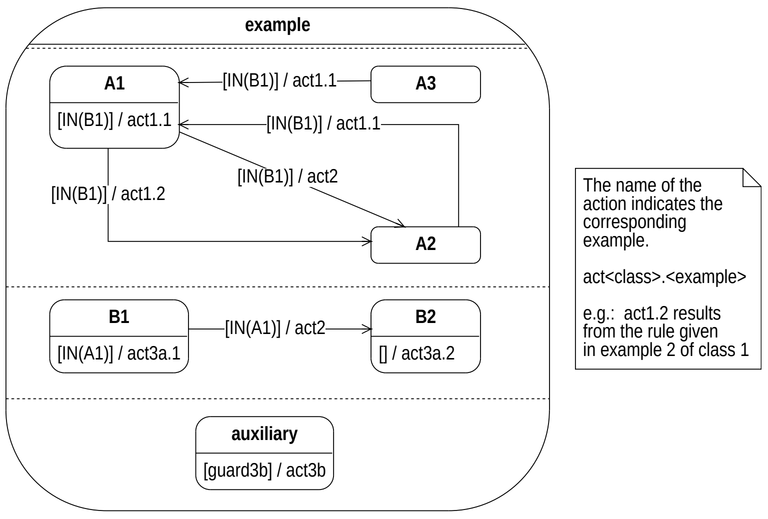


Figure 2: Classification of transitions

classification. The resulting transitions are also pictured in Figure 2. The name of the actions indicate the corresponding rule.

1. **Exactly one characterizing function is updated.** In this case, the transition can be associated with the orthogonal component corresponding to the changed characterizing function. Dependencies on other characterizing functions can be expressed via UML's **IN(***state***)** construct.
   e.g.:
   - **if** ch_b = B1 **then** act1.1, ch_a := A1
     $\Rightarrow * \times \mathbf{B1} \rightarrow \mathbf{A1} \times \mathbf{B1}$
     The value of the updated characterizing function (*ch_a*) is not restricted in the guard. Therefore, the translation results in three transitions with source state A1, A2, and A3 respectively and with destination state A1 (see Figure 2).
   - **if** ch_a = A1 **and** ch_b = B1 **then** act1.2, ch_a := A2
     $\Rightarrow \mathbf{A1} \times \mathbf{B1} \rightarrow \mathbf{A2} \times \mathbf{B1}$
     The value of the updated characterizing function (*ch_a*) is restricted to A1. Therefore, only one transition with source state A1 and destination state A2 is generated (see Figure 2).

   If *char_fun* is the updated characterizing function which has n possible values, then at most n transitions are generated.

2. **Several characterizing functions are updated.** In this case transitions in all orthogonal components corresponding to updated functions are generated. Other dependencies are treated as above. A simplification of the resulting transitions can be achieved by attaching the action to only one of the transitions, leaving the rest empty.
   e.g.:
   - **if** ch_a = A1 **and** ch_b = B1 **then** act2, ch_a := A2, ch_b := B2
     $\mathbf{A1} \times \mathbf{B1} \rightarrow \mathbf{A2} \times \mathbf{B2}$
     The value of both updated characterizing functions is restricted to A1, and B1 respectively. A transition from A1 to A2, and a transition from B1 to B2 is generated (see Figure 2).

If ch_i are the updated characterizing functions, each with n_i possible values, then at most n_i transitions are generated for each function.

3. **No characterizing function is updated.** This implies that no change of state occurs with respect to the STD.

   (a) **The transition depends on some characterizing Function.** In this case, the transition can be associated with any orthogonal component corresponding to one of these functions. The dependencies on other characterizing functions are expressed via UML's **IN(*state*)** construct. The choice of the component may be influenced by minimizing conflicting transitions (see section 3.2).
   e.g.:

   - **if** ch_a = A1 **and** ch_b = B1 **then** act3a.1
     $\Rightarrow \mathrm{A1} \times \mathrm{B1} \rightarrow \mathrm{A1} \times \mathrm{B1}$
     The source state of this transition is restricted to A1, and B1. The transition can be assigned to either A1 or B1 (see Figure 2).
   - **if** ch_b = B1 **then** act3a.2
     $\Rightarrow * \times \mathrm{B1} \rightarrow * \times \mathrm{B1}$
     Only *ch_b* is restricted. Therefore, the resulting transition should be assigned to B1 (see Figure 2).

   (b) **The transition does not depend on characterizing functions.** In this case, the transition cannot be associated with a single state.
   e.g.:
   if guard3b **then** act3b
   $\Rightarrow * \times * \rightarrow * \times *$
   No restrictions on characterizing functions are given. The resulting transition will be assigned to an 'auxiliary' state in a *new* orthogonal component (see Figure 2). All transitions which neither depend on, nor update any characterizing function then can be associated with this state.

iv) Resolving conflicting transitions

In ASMs, multiple rules can fire at once. In STDs, the firing of multiple transitions is allowed only, if these transitions are in separate orthogonal components. If no characterizing functions would be chosen, for each rule a separate component could be generated, therewith avoiding any conflicts.

The choice of characterizing functions now can force transitions originating from different rules to be placed in a common component. If such transitions have the same source state, and non exclusive guards, a conflict arises.

Due to the ASM concept of rules, the translation so far very likely results in such conflicting transitions. If the transitions cannot be put in different components, the conflicting transitions must be reformulated, such that (a) the resulting transitions are no longer in conflict, and (b) the action of the resulting transitions are the same as the the sum of the actions produced by the previously conflicting transitions under the same conditions.

E.g., assume two transitions t1 and t2.

- **if** guard_1 **then** action_1
- **if** guard_2 **then** action_2

If *guard_1*, and *guard_2* are not contradictory, then three new transitions must be generated as follows.

- **if** guard_1 **and** guard_2 **then** action_1, action_2
- **if** guard_1 **and not** guard_2 **then** action_1

- **if not** guard_1 **and** guard_2 **then** action_2

Doing the reformulation mechanically, the problem of absurd transitions, i.e. transitions never taken, arises. Even if in ASMs these transitions are implicitly present, too, there they do not produce syntactical overhead. In STDs, such transitions make diagrams confusing. And especially if orthogonal decomposition is not used, the syntactic overhead produced can be enormous.

Making the problem even worse, the action part of such transitions can contain contradicting updates. If the contradicting updates concern non characterizing functions, the problem is only semantic. But if the concerned functions are characterizing, a syntactic problem arises. The resulting transition would have two different destination states. This is syntactically not possible (there is just one arrow head).

In this case, it must be explicitly shown that such transitions are never taken, and appropriate guards must be added to the original rules. An example of such a situation is given in section 5.

3.3 Modules

Distributed ASMs

Modules are a feature of distributed ASMs. Each module represents an independent ASM communicating with the other modules via public functions. After initialization, the modules run asynchronously, i.e. firing of rules is not synchronized between modules. Nonetheless, the actions are still ensured to be atomic.

Each ASM module can be mapped into a UML class described through an associated STD. Objects of this class then run independently, providing the asynchronous behavior required by DASMs. A schema of the generated STDs is given in Figure 3.

To provide initialization, synchronization, and the global functions defined in the ASM, a additional 'system' STD is introduced. This diagram consists mainly of two states, one doing the initialization, and the other denoting the running system. Global functions are assigned with this 'system' class. For simplification, they can be used in other STDs without reference to the 'system' object, e.g. *System.sin(x)* can be referred to as *sin(x)*. This usage is unambiguous because there is only one 'system' object, and, due to the ASM origin, the function names are unique.

Each module is translated into a separate STD. On the root level, this STD consists of two states. The first state represents the need of initialization. It is left upon receiving an event signaling the end of the initialization in the 'system' object. This is necessary to synchronize the start of a run. The second state consists of the states, and transitions implied through the characterizing functions. Here, for the first time the concept of events provided by STDs is utilized.

Final states are not generated. This is due to the ASM semantic where no explicit finial states are given either.

Non-distributed ASMs

Non distributed ASMs can be seen as distributed ASMs consisting of just one module. Naturally, there is no distinction between global and local function.

The 'system' state can be skipped. All the initialization can be done in the local initialization state. Synchronization is not necessary.

A schema of the STD for a non-distributed ASM is given in figure 4.

4 Case study: Prolog machine (1st version)

The following ASM is taken from [BR94], section 1.1. It defines an operational semantics of Prolog. We do not explain the ASM here. Also, we do not give the full rules, as

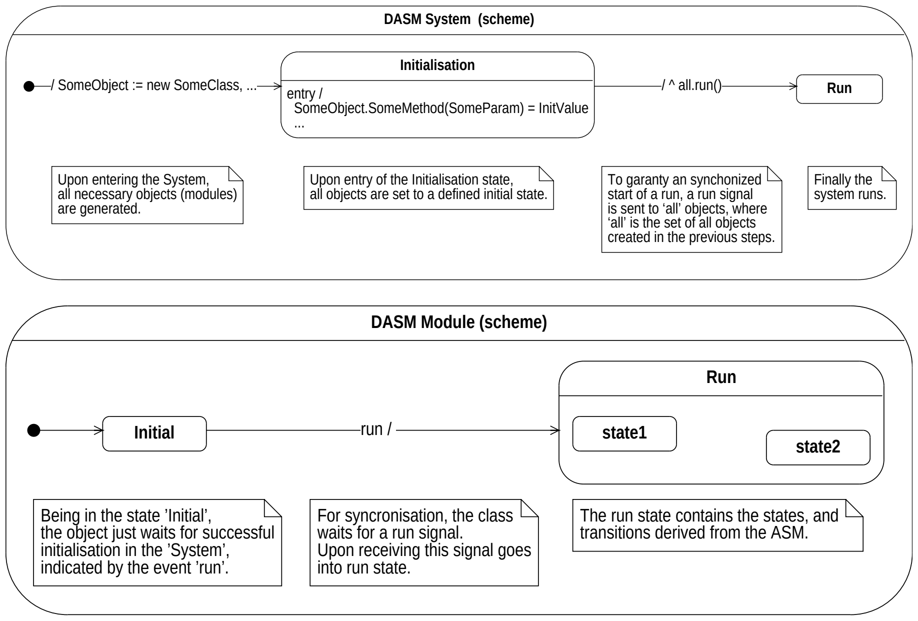

DASM System (scheme)
/ SomeObject := new SomeClass, ...
Initialisation
entry /
SomeObject.SomeMethod(SomeParam) = InitValue
...
/ ^ all.run()
Run
Upon entering the System, all necessary objects (modules) are generated.
Upon entry of the Initialisation state, all objects are set to a defined initial state.
To garanty an synchonized start of a run, a run signal is sent to 'all' objects, where 'all' is the set of all objects created in the previous steps.
Finally the system runs.
DASM Module (scheme)
Initial
run /
Run
state1
state2
Being in the state 'Initial', the object just waits for successful initialisation in the 'System', indicated by the event 'run'.
For syncronisation, the class waits for a run signal. Upon receiving this signal goes into run state.
The run state contains the states, and transitions derived from the ASM.

Figure 3: Schema for DASM system, and module diagram

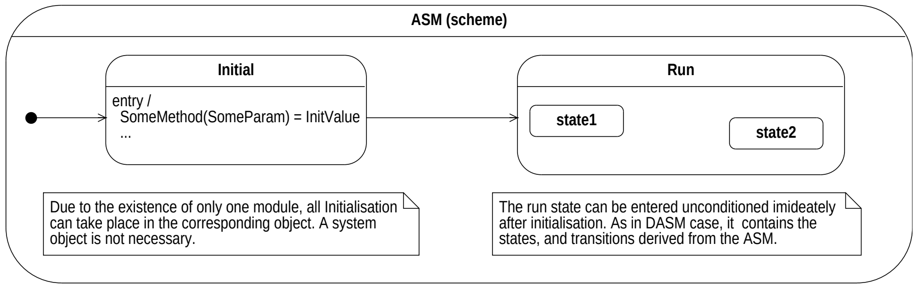

ASM (scheme)
Initial
entry /
SomeMethod(SomeParam) = InitValue
...
Run
state1
state2
Due to the existence of only one module, all Initialisation can take place in the corresponding object. A system object is not necessary.
The run state can be entered unconditioned imideately after initialisation. As in DASM case, it contains the states, and transitions derived from the ASM.

Figure 4: Schema for a non distributed ASM

some abbreviations are not expanded. The form presented below suffices to discuss the translation into an STD.

The following ASM is taken from an ASM specification of a Prolog machine as given in [BR94] section 1.1. It defines an operational semantics of Prolog.

Characterizing functions

As characterizing functions *mode : {Select,Call}*, and *stop : {Running,Success,Fail}* are chosen. Therefore, the states are given through $\{Select, Call\} \times \{Running, Success, Fail\}$.

Flat transition rules

As in the original ASM specification, all transition rules implicitly have an additional guard *stop = running*. For clarity of the following translation, the equalities, and updates involving a characterizing function are emphasized.

```
1.
    if     decglesq = empty_list
    then   stop := Success
2.
    if     is_user_defined(fst(goal)) and mode = Call
    then   extend NODE by ...
           mode := Select
3.
    if     fst(fst(decglseq)) = empty_list
    then   decglseq := rest(decglseq)
4.
    if     is_user_defined(fst(goal)) and mode = Select and
           cands = empty_list and father = root
    then   mode := Fail
5.
    if     is_user_defined(fst(goal)) and mode = Select and
           cands = empty_list and not father = root
    then   currendnode := father
           mode := Select
6.
    if     is_user_defined(fst(goal)) and mode = Select and
           not cands = empty_list and Phi = nil
    then   cands:= rest(cands)
7.
    if     is_user_defined(fst(goal)) and mode = Select and
           not cands = empty_list and not Phi = nil
    then   currnode := father
           cands := rest(cands)
           mode := Call
           decglseq(fst(cands)) := new_decglseq
           vi := vi + 1
8.
    if     fst(goal) = true
    then   decglseq := cont
9.
    if     fst(goal) = fail and father = root
    then   stop := Fail
```

10.
```
if      fst(goal) = fail and not father = root
then    currendnode := father
        mode := Select
```
11.
```
if      fst(goal) = cut
then    father := cutpt
        decglseq := cont
```

The following table shows the profile and classification of each rule. Also, the source states of the resulting transitions are given.

rule	profile	class.	associated with
1.	$* \times$ Running $\rightarrow * \times$ Running	3(a)	Running
2.	Call $\times$ Running $\rightarrow$ Select $\times$ Running	1	Call
3.	$* \times$ Running $\rightarrow * \times$ Running	3(a)	Running
4.	Select $\times$ Running $\rightarrow$ Select $\times$ Fail	1	Running
5.	Select $\times$ Running $\rightarrow$ Select $\times$ Running	3(a)	Select **or** Running
6.	Select $\times$ Running $\rightarrow$ Select $\times$ Running	3(a)	Select **or** Running
7.	Select $\times$ Running $\rightarrow$ Call $\times$ Running	1	Select
8.	$* \times$ Running $\rightarrow * \times$ Running	3(a)	Running
9.	$* \times$ Running $\rightarrow * \times$ Fail	1	Running
10.	$* \times$ Running $\rightarrow$ Select $\times$ Running	1	Select **and** Call
11.	$* \times$ Running $\rightarrow * \times$ Running	3(a)	Running

Fortunately, there are no conflicting transitions to be resolved. The result of the transformation is given in Figure 5. To demonstrate the usefulness of the exploited orthogonal decomposition, a translation *not using* orthogonal decomposition is given in Figure 6.

For visualization, Cayenne's CASE tool **ObjectTeam** is used. Unfortunately, **ObjectTeam** sets some restrictions on the syntax of transitions. Principally, this restrictions are not given in UML. To circumvent this restrictions, the concerned statements are placed in UML comment boxes. Within the transition, an *eval()* refers to the current statement applying. To make this fully UML conform, it would have to be defined as stereotype.

5 Case study: Prolog machine (2nd version)

The following ASM is a refinement of the previous. It is also taken from [BR94], described in section 1.3. Now, the *Select* state of the previous example is split in to three new states.

Characterizing functions

The characterizing functions are chosen as above, but the image of mode now is {*Call*, *Enter*, *Try*, *Retry*}. The functionality of mode *Select* is split now over the modes *Enter*, *Try*, and *Retry*.

Flat transitions rules

As in the original ASM specification, all transition rules implicitly have an additional guard *stop = running*. Again, the equalities and updates involving a characterizing function are emphasized.

1.
```
if      decglesq = empty_list
then    stop := Success
```

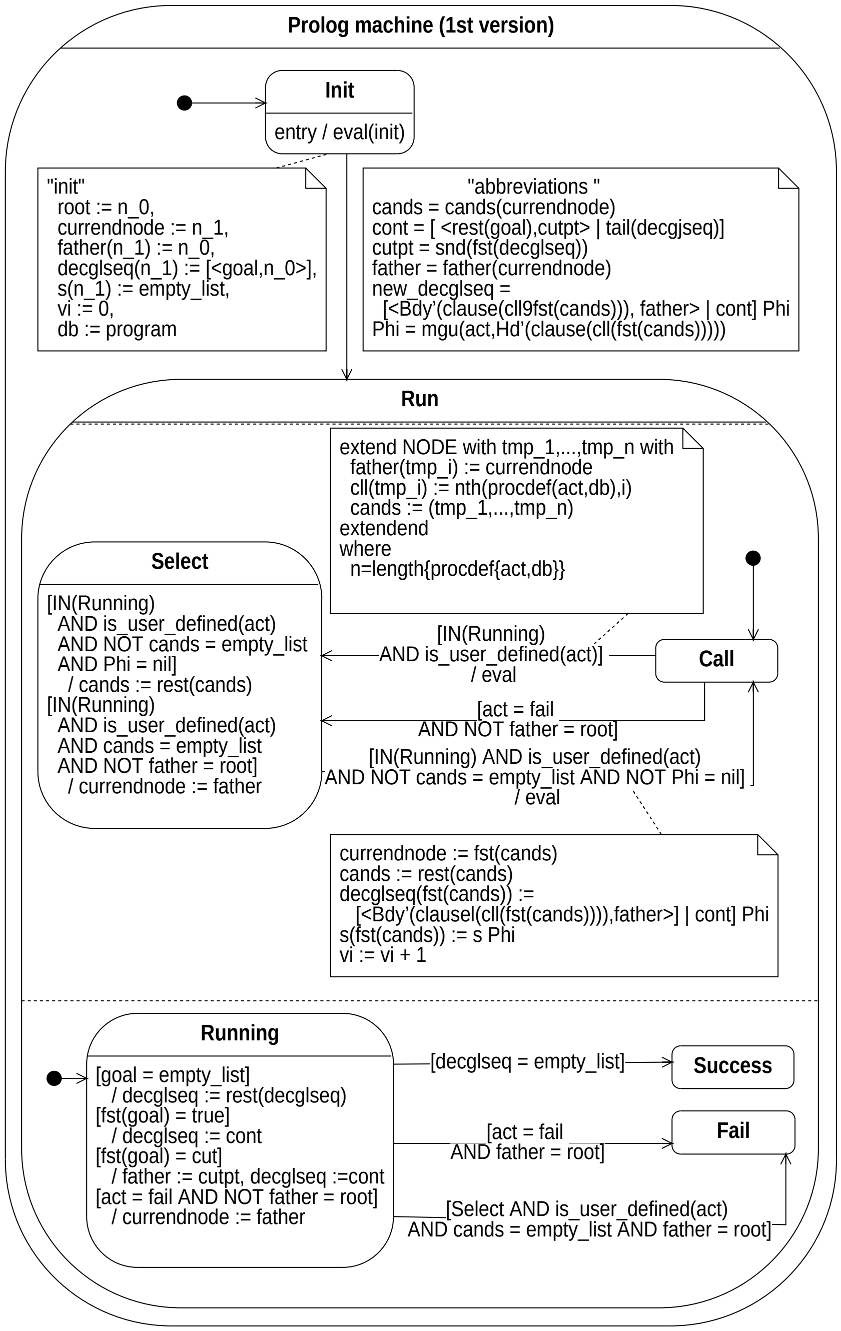

Prolog machine (1st version)
Init
entry / eval(init)
"init"
root := n_0,
currendnode := n_1,
father(n_1) := n_0,
decglseq(n_1) := [<goal,n_0>],
s(n_1) := empty_list,
vi := 0,
db := program
"abbreviations "
cands = cands(currendnode)
cont = [<rest(goal),cutpt> | tail(decgjseq)]
cutpt = snd(fst(decglseq))
father = father(currendnode)
new_decglseq =
[<Bdy'(clause(cll9fst(cands))), father> | cont] Phi
Phi = mgu(act,Hd'(clause(cll(fst(cands)))))
Run
extend NODE with tmp_1,...,tmp_n with
father(tmp_i) := currendnode
cll(tmp_i) := nth(procdef(act,db),i)
cands := (tmp_1,...,tmp_n)
extendend
where
n=length{procdef{act,db}}
Select
[IN(Running)
AND is_user_defined(act)
AND NOT cands = empty_list
AND Phi = nil]
/ cands := rest(cands)
[IN(Running)
AND is_user_defined(act)
AND cands = empty_list
AND NOT father = root]
/ currendnode := father
[IN(Running)
AND is_user_defined(act)]
/ eval
Call
[act = fail
AND NOT father = root]
[IN(Running) AND is_user_defined(act)
AND NOT cands = empty_list AND NOT Phi = nil]
/ eval
currendnode := fst(cands)
cands := rest(cands)
decglseq(fst(cands)) :=
[<Bdy'(clausel(cll(fst(cands)))),father>] | cont] Phi
s(fst(cands)) := s Phi
vi := vi + 1
Running
[goal = empty_list]
/ decglseq := rest(decglseq)
[fst(goal) = true]
/ decglseq := cont
[fst(goal) = cut]
/ father := cutpt, decglseq :=cont
[act = fail AND NOT father = root]
/ currendnode := father
[decglseq = empty_list]
Success
[act = fail
AND father = root]
Fail
[Select AND is_user_defined(act)
AND cands = empty_list AND father = root]

Figure 5: STD for Prolog machine (1st version, orthogonal product)

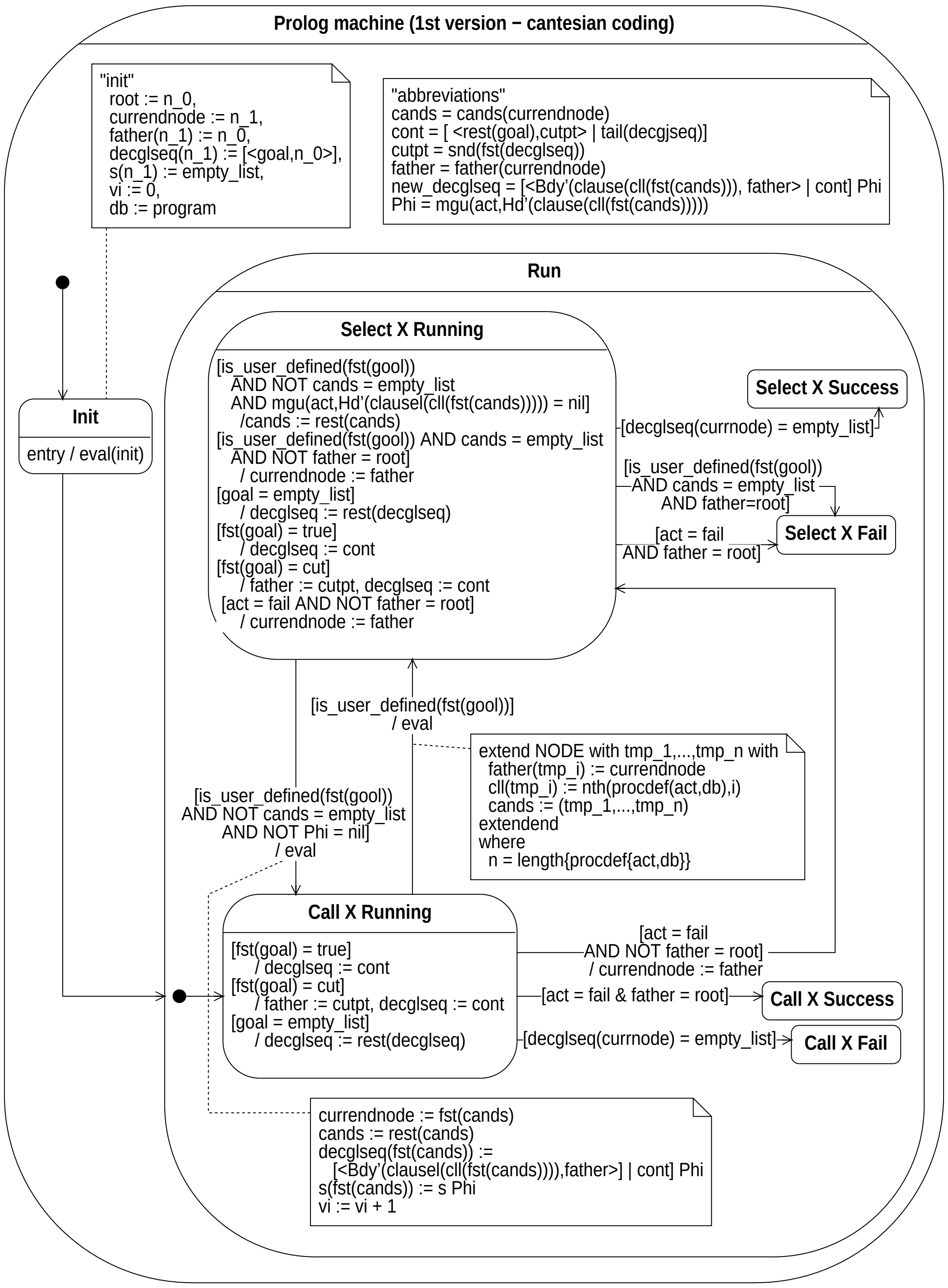

Prolog machine (1st version − cantesian coding)
"init"
root := n_0,
currendnode := n_1,
father(n_1) := n_0,
decglseq(n_1) := [<goal,n_0>],
s(n_1) := empty_list,
vi := 0,
db := program
"abbreviations"
cands = cands(currendnode)
cont = [<rest(goal),cutpt> | tail(decgjseq)]
cutpt = snd(fst(decglseq))
father = father(currendnode)
new_decglseq = [<Bdy'(clause(cll(fst(cands))), father> | cont] Phi
Phi = mgu(act,Hd'(clause(cll(fst(cands)))))
Init
entry / eval(init)
Run
Select X Running
[is_user_defined(fst(gool))
AND NOT cands = empty_list
AND mgu(act,Hd'(clausel(cll(fst(cands))))) = nil]
/cands := rest(cands)
[is_user_defined(fst(gool)) AND cands = empty_list
AND NOT father = root]
/ currendnode := father
[goal = empty_list]
/ decglseq := rest(decglseq)
[fst(goal) = true]
/ decglseq := cont
[fst(goal) = cut]
/ father := cutpt, decglseq := cont
[act = fail AND NOT father = root]
/ currendnode := father
Select X Success
[decglseq(currnode) = empty_list]
[is_user_defined(fst(gool))
AND cands = empty_list
AND father=root]
Select X Fail
[act = fail
AND father = root]
[is_user_defined(fst(gool))]
/ eval
extend NODE with tmp_1,...,tmp_n with
father(tmp_i) := currendnode
cll(tmp_i) := nth(procdef(act,db),i)
cands := (tmp_1,...,tmp_n)
extendend
where
n = length{procdef{act,db}}
[is_user_defined(fst(gool))
AND NOT cands = empty_list
AND NOT Phi = nil]
/ eval
Call X Running
[fst(goal) = true]
/ decglseq := cont
[fst(goal) = cut]
/ father := cutpt, decglseq := cont
[goal = empty_list]
/ decglseq := rest(decglseq)
[act = fail
AND NOT father = root]
/ currendnode := father
[act = fail & father = root]
Call X Success
[decglseq(currnode) = empty_list]
Call X Fail
currendnode := fst(cands)
cands := rest(cands)
decglseq(fst(cands)) :=
[<Bdy'(clausel(cll(fst(cands)))),father>] | cont] Phi
s(fst(cands)) := s Phi
vi := vi + 1

Figure 6: STD for Prolog machine (1st version, direct product)

```
2.
    if      is_user_defined(fst(goal)) and  mode = Call and
            clause(procdef(act,db)) = nil and b = bottom
    then    stop := Fail
3.
    if      is_user_defined(fst(goal)) and mode = Call and
            clause(procdef(act,db)) = nil and not b = bottom
    then    mode := Retry
4.
    if      is_user_defined(fst(goal)) and mode = Call and
            not clause(procdef(act,db)) = nil
    then    cll := procdef(act,db))
            mode := Try
            ct := b
5.
    if      goal = empty_list
    then    decglseq := rest(decglseq)
6.
    if      act = true
    then    decglseq := cont
7.
    if      act = fail and b(b) = bottom
    then    stop := Fail
8.
    if      act = fail and not b(b) = bottom
    then    b := b(b)
9.
    if      act = cut
    then    b := cutpt
            decglseq := cont
10.
    if      mode = Try
    then    push tmp with . . .
            mode := Enter
11.
    if      mode = Enter and Phi = nil and b = bottom
    then    stop := Fail
12.
    if      mode = Enter and Phi = nil and not b = bottom
    then    mode := Retry
13.
    if      mode = Enter and not Phi = nil
    then    decglseq := new_decglseq
            s := s Phi
            vi := vi + 1
            mode := Call
14.
    if      mode = Retry and clause(cll(b)) = nil and
            b(b) = bottom
    then    stop := Fail
15.
    if      mode = Retry and clause(cll(b)) = nil and
            not b(b) = bottom
    then    b := b(b)
```

16.
```
if    $mode = Retry$ and not clause(cll(b)) = nil
then  fetch_state_from(b)
      cll := cll(b)
      cll(b) := cll(b)+
      ct := b(b)
      $mode := Enter$
```

The following table shows the profile, and classification of each rule. Also, the source states of the resulting transitions are given.

rule	profile	class.	associated with
1.	$* \times$ Running $\to * \times$ Success	1	Running
2.	Call $\times$ Running $\to$ Call $\times$ Fail	1	Running
3.	Call $\times$ Running $\to$ Retry $\times$ Running	1	Call
4.	Call $\times$ Running $\to$ Try $\times$ Running	1	Call
5.	$* \times$ Running $\to * \times$ Running	3(a)	Running
6.	$* \times$ Running $\to * \times$ Running	3(a)	Running
7.	$* \times$ Running $\to * \times$ Fail	1	Running
8.	$* \times$ Running $\to * \times$ Running	3(a)	Running
9.	$* \times$ Running $\to * \times$ Running	3(a)	Running
10.	Try $\times$ Running $\to$ Enter $\times$ Running	1	Try
11.	Enter $\times$ Running $\to$ Enter $\times$ Fail	1	Running
12.	Enter $\times$ Running $\to$ Retry $\times$ Running	1	Enter
13.	Enter $\times$ Running $\to$ Call $\times$ Running	1	Enter
14.	Retry $\times$ Running $\to$ Retry $\times$ Fail	1	Running
15.	Retry $\times$ Running $\to$ Retry $\times$ Running	3(a)	Retry **or** Running
16.	Retry $\times$ Running $\to$ Enter $\times$ Running	1	Retry

With this mapping, the problem of confliction transitions arrises. Rules 1, 5, 6, 7, 8, and 9 are in conflict with rules 11, and 14.

A first step to resolve the conflicts could be to place rules 5, 6, 8, and 9 in an auxiliary state in a new orthogonal component. This can be done because these rules do not change characterizing functions.

Now, rules 7, 11, and 14 can be placed into a single transition, combining their guards via **or** . This can be done due to the fact that they have the same destination state, and the same action (namely non).

The only conflict left is the conflict of rule 1 with rule 11, and 14. Unfotunately, these rules can not be combined, because they have contradicting updates (rule 1: *stop := success*, rule 11, 14: *stop := Fail*). To resolve this conflict, it must be shown, that rule 1 is never triggered together with rule 11, or 14.

As mentioned in [BR94] section 1.1, it can even be shown that rules 1, 5, 6, 7, 8, and 9 are only triggered in Call mode. Including this infomation in the rules (by placeing *mode = Call* as an additional guard in these rules) results in the following classification of rules.

rule	profile	class.	associated with
1.	Call × Running → Call × Success	1	Running
2.	Call × Running → Call × Fail	1	Running
3.	Call × Running → Retry × Running	1	Call
4.	Call × Running → Try × Running	1	Call
5.	Call × Running → Call × Running	3(a)	Running
6.	Call × Running → Call × Running	3(a)	Running
7.	Call × Running → Call × Fail	1	Running
8.	Call × Running → Call × Running	3(a)	Running
9.	Call × Running → Call × Running	3(a)	Running
10.	Try × Running → Enter × Running	1	Try
11.	Enter × Running → Enter × Fail	1	Running
12.	Enter × Running → Retry × Running	1	Enter
13.	Enter × Running → Call × Running	1	Enter
14.	Retry × Running → Retry × Fail	1	Running
15.	Retry × Running → Retry × Running	3(a)	Retry **or** Running
16.	Retry × Running → Enter × Running	1	Retry

Using the new rules, there are no more conflicts. The result of the transformation is given in Figure 7. Again, a translation not using orthogonal decomposition is given, see Figure 8.

6 Case study: Alternating Bit Protocol (asymetric version)

In this example, an ASM specification of an alternating bit protocol is translated into a STD. The ASM specification is taken from [Hug95], section 2. The ASM does not contain nested conditions. The resulting STD is given in Figures 9 to 12.

Characterizing functions

As characterizing functions the two functions *SenderBit : bits* and *ReceiverBit : bits* are chosen for the sender and receiver module, respectively. For the other modules, no characterizing function is chosen.

Classified transitions

In this example, the choice of characterizing functions and the specific guards results in a duplication of rules (e.g.: 'Bit(SenderInMsg) = SenderBit' generates 'Bit(SenderInMsg) = SenderBit = 0', and 'Bit(SenderInMsg) = SenderBit = 1').

Module: Sender

Transitions resulting from ProcessAck, Timeout, and ClearMessage are in conflict. But, ReTransmit, and ClearMessage are independent from the current state. Therefore, the conflict can be resolved by putting them in orthogonal components.

```
Rule: ReTransmit (* → *)
 if     Timeout
 then   ReceiverQueue := ReceiverQueue
        ++ Msg(SenderFile(SenderNo),SenderBit)
        Timeout := false
Rule: ProcessAck (SenderBit=0 → SenderBit=1)
```

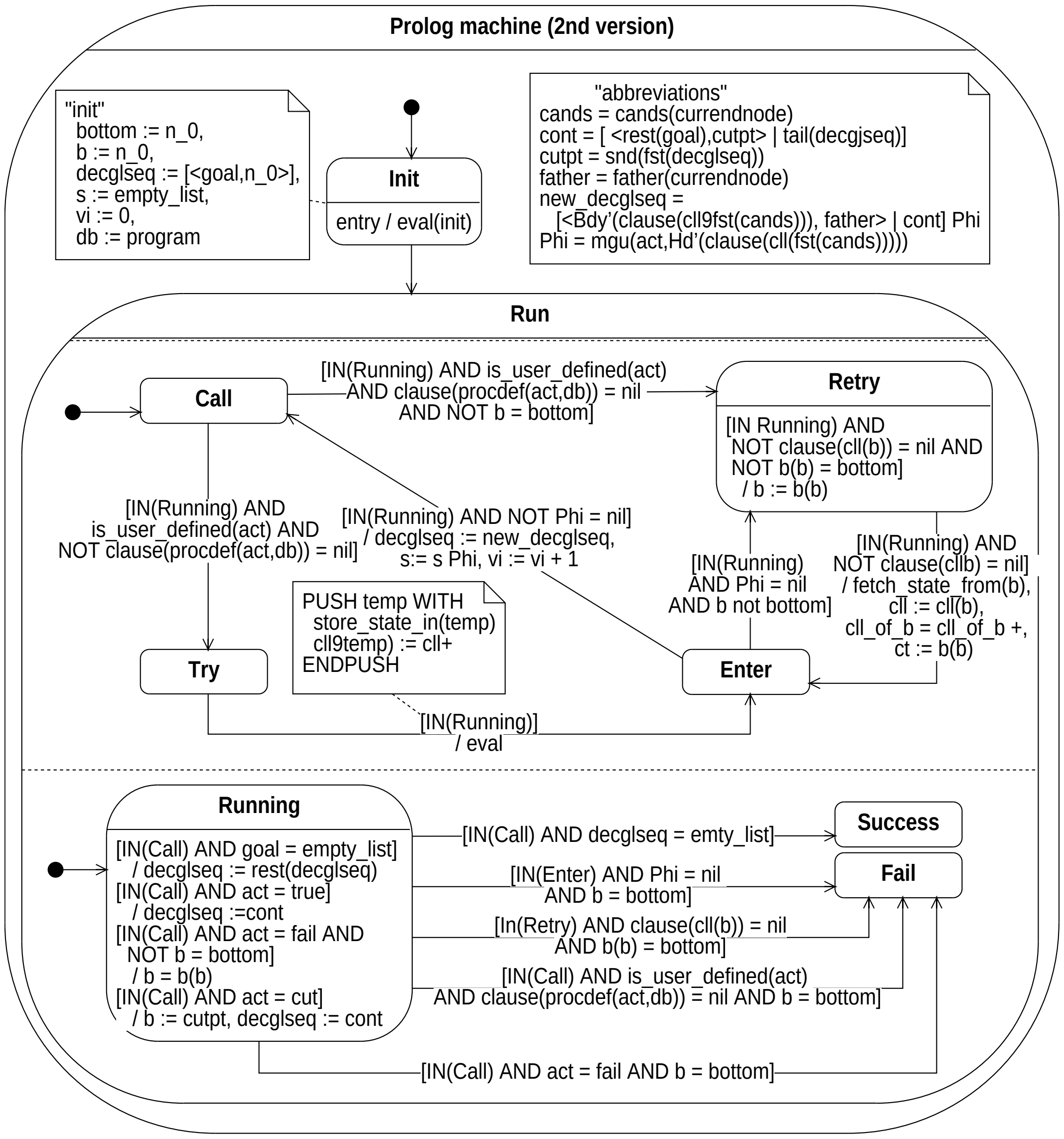

Prolog machine (2nd version)
"init"
bottom := n_0,
b := n_0,
decglseq := [<goal,n_0>],
s := empty_list,
vi := 0,
db := program
Init
entry / eval(init)
"abbreviations"
cands = cands(currendnode)
cont = [<rest(goal),cutpt> | tail(decgjseq)]
cutpt = snd(fst(decglseq))
father = father(currendnode)
new_decglseq =
[<Bdy'(clause(cll9fst(cands))), father> | cont] Phi
Phi = mgu(act,Hd'(clause(cll(fst(cands)))))
Run
Call
[IN(Running) AND is_user_defined(act) AND clause(procdef(act,db)) = nil AND NOT b = bottom]
Retry
[IN Running) AND NOT clause(cll(b)) = nil AND NOT b(b) = bottom] / b := b(b)
[IN(Running) AND is_user_defined(act) AND NOT clause(procdef(act,db)) = nil]
[IN(Running) AND NOT Phi = nil] / decglseq := new_decglseq, s:= s Phi, vi := vi + 1
[IN(Running) AND Phi = nil AND b not bottom]
[IN(Running) AND NOT clause(cllb) = nil] / fetch_state_from(b), cll := cll(b), cll_of_b = cll_of_b +, ct := b(b)
PUSH temp WITH store_state_in(temp) cll9temp) := cll+ ENDPUSH
Try
Enter
[IN(Running)] / eval
Running
[IN(Call) AND goal = empty_list] / decglseq := rest(decglseq)
[IN(Call) AND act = true] / decglseq :=cont
[IN(Call) AND act = fail AND NOT b = bottom] / b = b(b)
[IN(Call) AND act = cut] / b := cutpt, decglseq := cont
[IN(Call) AND decglseq = emty_list]
Success
Fail
[IN(Enter) AND Phi = nil AND b = bottom]
[In(Retry) AND clause(cll(b)) = nil AND b(b) = bottom]
[IN(Call) AND is_user_defined(act) AND clause(procdef(act,db)) = nil AND b = bottom]
[IN(Call) AND act = fail AND b = bottom]

Figure 7: STD for Prolog machine (2st version, orthogonal product)

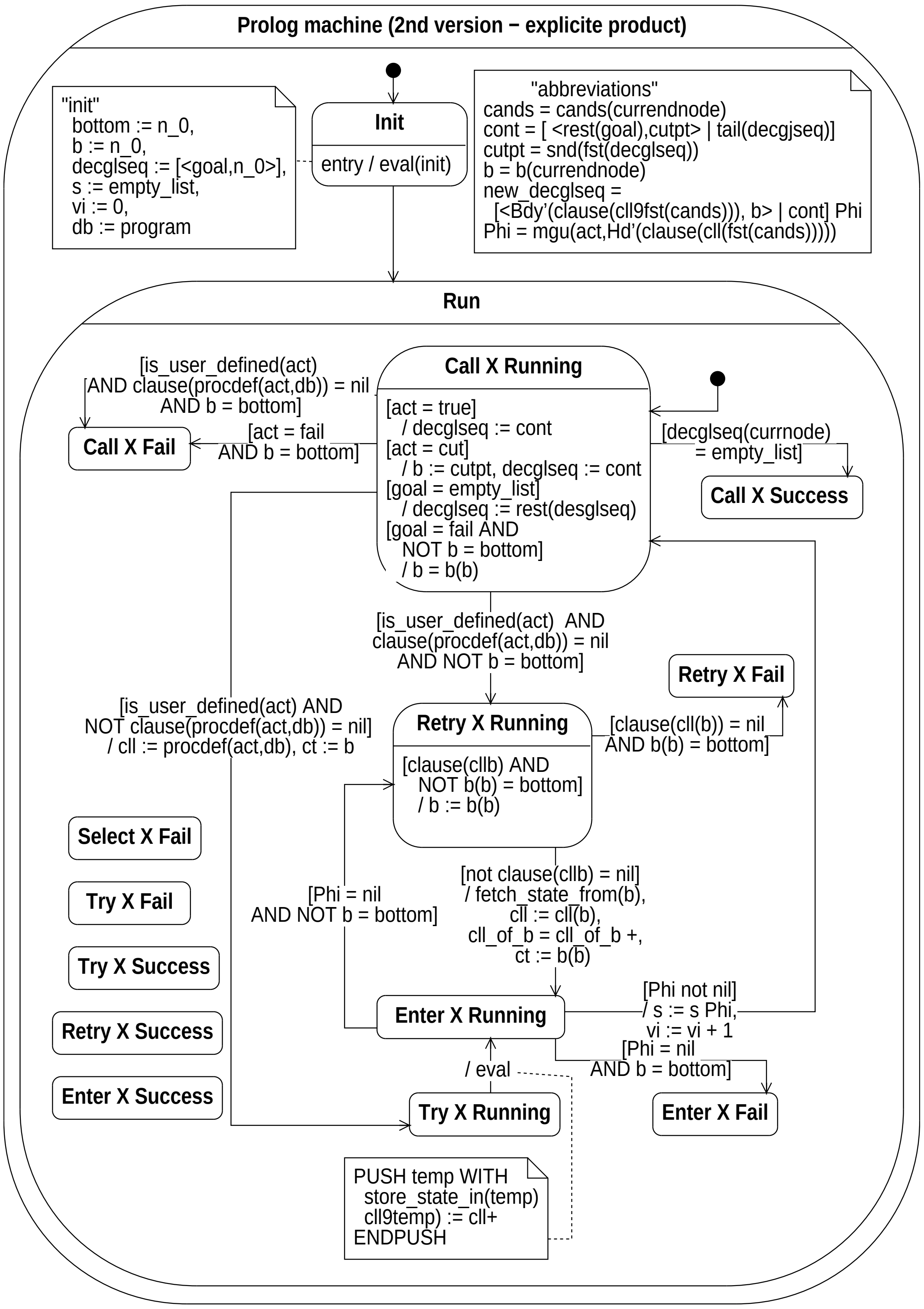


Figure 8: STD for Prolog machine (2st version, direct product)

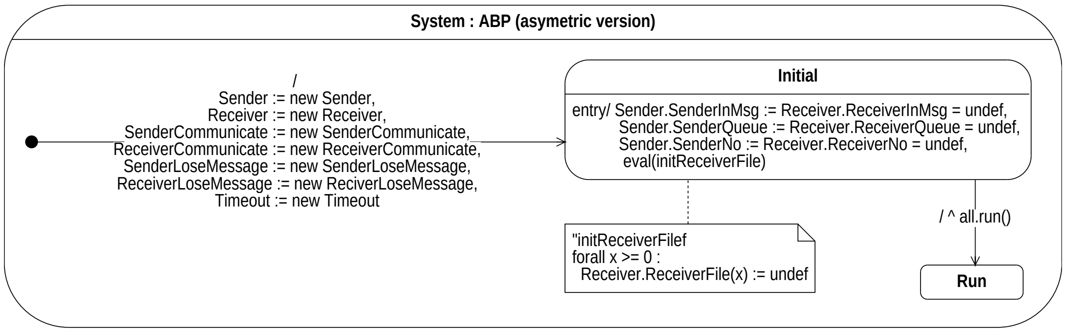

System : ABP (asymetric version)
/
Sender := new Sender,
Receiver := new Receiver,
SenderCommunicate := new SenderCommunicate,
ReceiverCommunicate := new ReceiverCommunicate,
SenderLoseMessage := new SenderLoseMessage,
ReceiverLoseMessage := new ReciverLoseMessage,
Timeout := new Timeout
Initial
entry/ Sender.SenderInMsg := Receiver.ReceiverInMsg = undef,
Sender.SenderQueue := Receiver.ReceiverQueue = undef,
Sender.SenderNo := Receiver.ReceiverNo = undef,
eval(initReceiverFile)
"initReceiverFilef
forall x >= 0 :
Receiver.ReceiverFile(x) := undef
/ ^ all.run()
Run

Figure 9: STD for ABP System, asymmetric version

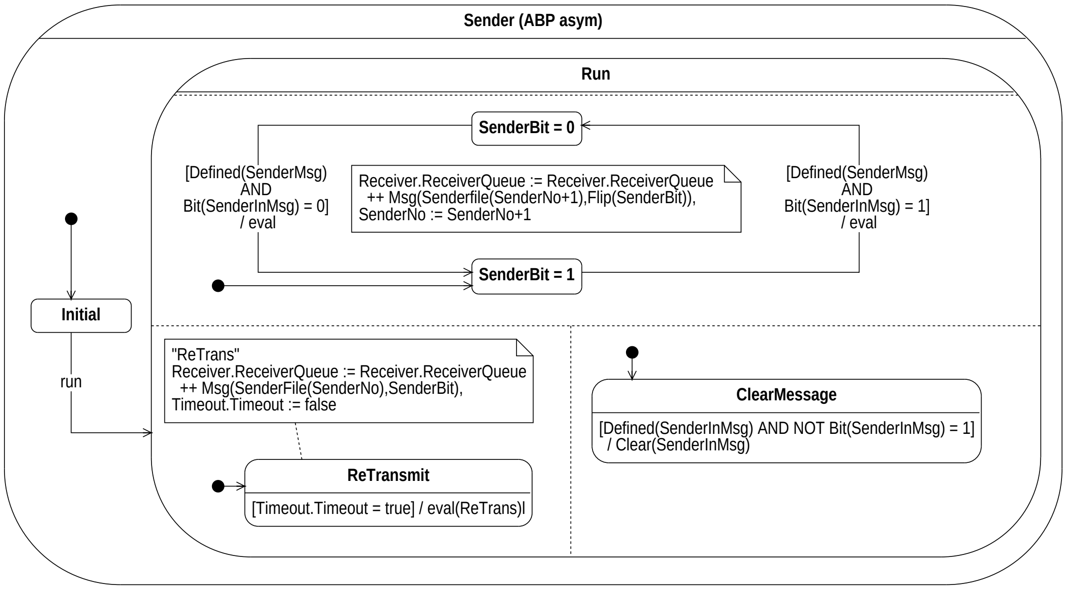
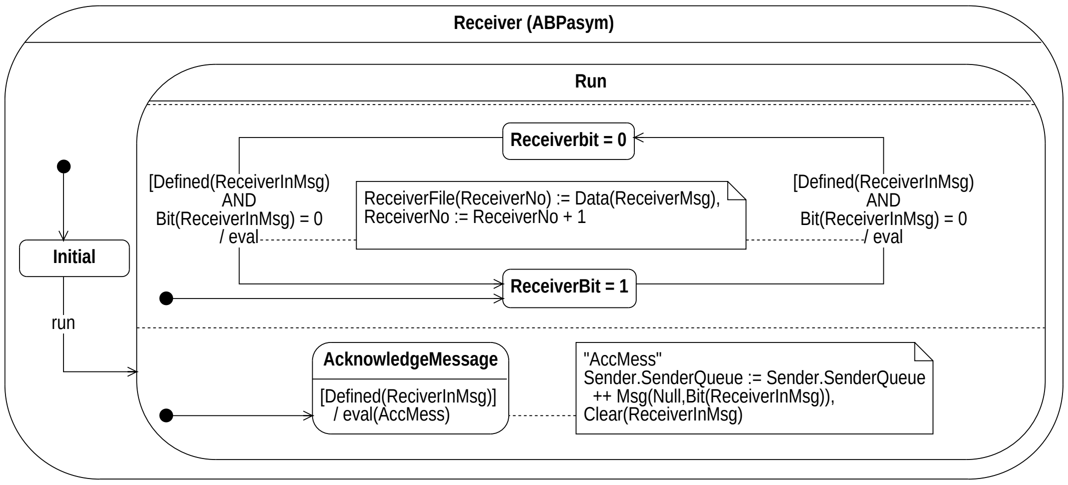
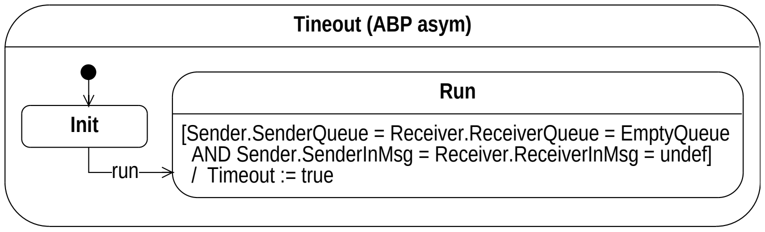

Sender (ABP asym)
Run
SenderBit = 0
[Defined(SenderMsg)
AND
Bit(SenderInMsg) = 0]
/ eval
Receiver.ReceiverQueue := Receiver.ReceiverQueue
++ Msg(Senderfile(SenderNo+1),Flip(SenderBit)),
SenderNo := SenderNo+1
[Defined(SenderMsg)
AND
Bit(SenderInMsg) = 1]
/ eval
SenderBit = 1
Initial
run
"ReTrans"
Receiver.ReceiverQueue := Receiver.ReceiverQueue
++ Msg(SenderFile(SenderNo),SenderBit),
Timeout.Timeout := false
ReTransmit
[Timeout.Timeout = true] / eval(ReTrans)l
ClearMessage
[Defined(SenderInMsg) AND NOT Bit(SenderInMsg) = 1]
/ Clear(SenderInMsg)

Figure 10: STD for ABP Sender, asymmetric version



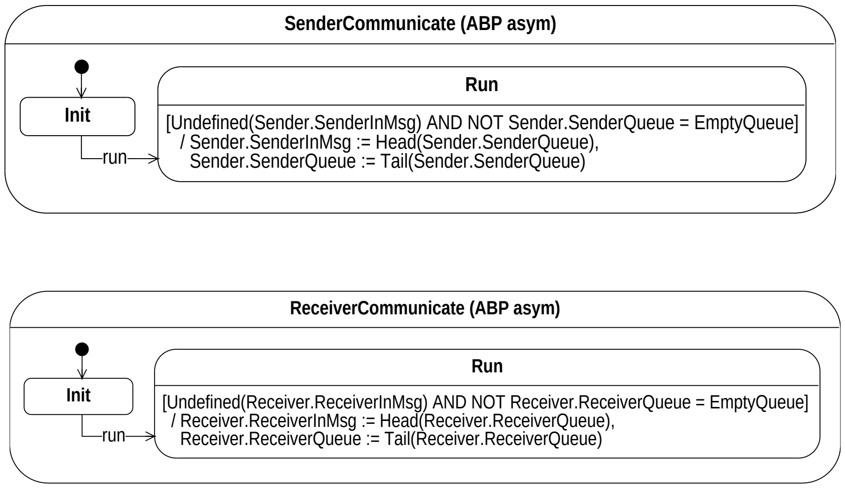
Receiver (ABPasym)
Run
Initial
run
Receiverbit = 0
ReceiverBit = 1
[Defined(ReceiverInMsg) AND Bit(ReceiverInMsg) = 0 / eval
ReceiverFile(ReceiverNo) := Data(ReceiverMsg), ReceiverNo := ReceiverNo + 1
[Defined(ReceiverInMsg) AND Bit(ReceiverInMsg) = 0 / eval
AcknowledgeMessage
[Defined(ReciverInMsg)] / eval(AccMess)
"AccMess" Sender.SenderQueue := Sender.SenderQueue ++ Msg(Null,Bit(ReceiverInMsg)), Clear(ReceiverInMsg)

Figure 11: STD for ABP Receiver, asymmetric version



Tineout (ABP asym)
Init
run
Run
[Sender.SenderQueue = Receiver.ReceiverQueue = EmptyQueue AND Sender.SenderInMsg = Receiver.ReceiverInMsg = undef] / Timeout := true

Figure 12: STD for ABP Timeout, asymmetric version



SenderCommunicate (ABP asym)
Init
run
Run
[Undefined(Sender.SenderInMsg) AND NOT Sender.SenderQueue = EmptyQueue] / Sender.SenderInMsg := Head(Sender.SenderQueue), Sender.SenderQueue := Tail(Sender.SenderQueue)
ReceiverCommunicate (ABP asym)
Init
run
Run
[Undefined(Receiver.ReceiverInMsg) AND NOT Receiver.ReceiverQueue = EmptyQueue] / Receiver.ReceiverInMsg := Head(Receiver.ReceiverQueue), Receiver.ReceiverQueue := Tail(Receiver.ReceiverQueue)

Figure 13: STD for ABP Communicate, asymmetric version

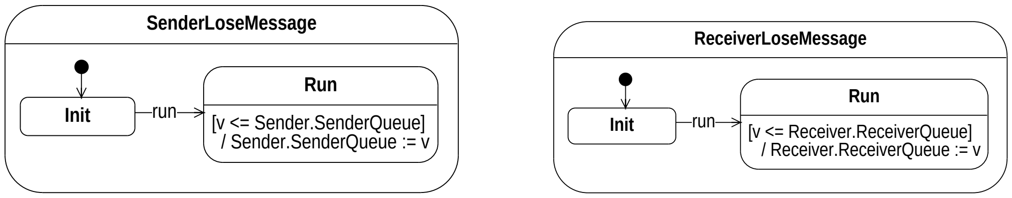

SenderLoseMessage
Init
run
Run
[v <= Sender.SenderQueue]
/ Sender.SenderQueue := v
ReceiverLoseMessage
Init
run
Run
[v <= Receiver.ReceiverQueue]
/ Receiver.ReceiverQueue := v

Figure 14: STD for ABP LoseMessage, asymmetric version

```
 if     Defined(SenderInMsg) and Bit(SenderInMsg)=SenderBit=0
 then   ReceiverQueue := ReceiverQueue
        ++ Msg(SenderFile(SenderNo),Flip(SenderBit))
        SenderBit := Flip(SenderBit)
        SenderNo := SenderNo + 1
Rule: ProcessAck (SenderBit=1 → SenderBit=0)
 if     Defined(SenderInMsg) and Bit(SenderInMsg)=SenderBit=1
 then   ReceiverQueue := ReceiverQueue
        ++ Msg(SenderFile(SenderNo),Flip(SenderBit))
        SenderBit := Flip(SenderBit)
        SenderNo := SenderNo + 1
Rule: ClearMessage (∗ → ∗)
 if     Defined(SenderInMsg)
 then   Clear(SenderInMsg)
```

Module: Receiver

Here, AcceptDatum and AcknowledgeMessage are in conflict. This conflict can be resolved similar to the sender module.

```
Rule: AcceptDatum (ReceiverBit=0 → ReceiverBit=1)
 if     Defined(ReceiverInMsg) and
        Bit(ReceiverInMsg)=ReceiverBit=0
 then   ReceiverFile(ReceiverNo := Data(ReceiverInMsg)
        ReceiverNo := ReceiverNo + 1
        ReceiverBit := Flip(ReceiverBit)
Rule: AcceptDatum (ReceiverBit=1 → ReceiverBit=0)
 if     Defined(ReceiverInMsg) and
        Bit(ReceiverInMsg)=ReceiverBit=1
 then   ReceiverFile(ReceiverNo := Data(ReceiverInMsg)
        ReceiverNo := ReceiverNo + 1
        ReceiverBit := Flip(ReceiverBit)
Rule: AcknowledgeMessage (∗ → ∗)
 if     Defined(ReceiverInMsg)
 then   SenderQueue:= SenderQueue
        ++ Msg(Null,Bit(ReceiverInMsg))
        Clear(ReceiverInMsg)
```

Module: SenderCommunicate

```
 if     Undefined(SenderInMsg) and
        not SenderQueue = EmptyQueue
 then   SenderInMsg := Head(SenderQueue)
        SenderQueue := Tail(SenderQueue)
```

Module: ReceiverCommunicate

```
if      Undefined(ReceiverInMsg) and
        not ReceiverQueue = EmptyQueue
then    ReceiverInMsg := Head(ReceiverQueue)
        ReceiverQueue := Tail(ReceiverQueue)
```

Module: SenderLoseMessage

```
if      v ≤ SenderQueue
then    SenderQueue := v
```

Module: ReceiverLoseMessage

```
if      v ≤ ReceiverQueue
then    ReceiverQueue := v
```

Module: Timeout

```
if      SenderQueue = ReceiverQueue = EmptyQueue and
        SenderInMsg = ReceiverInMsg = undef
then    Timeout := true
```

7 Case study: Alternating Bit Protocol (symmetric version)

This example is a symmetric version of the previous ASM. It is taken from [Hug95], section 3. The modules for sender, and rceiver are specified symetricaly. Sender, and receiver are disinguished using the fuction *"Me"*. The resulting STD is given in Figure 15 to 18.

Characterizing functions

As characterizing functions for the Sender/Receiver Module the functions *mode : {Put,Get}* is chosen. For the other modules no characterizing function is chosen.

Classified transitions

Module: Sender/Receiver Template

```
if      Mode = Put and Me = Sender
then    Q(You) := Q(You)
        ++ Msg(File(MyNum+1),(MyNum+1)mod 2)
        LastMsg := Msg(File(MyNum+1,(MyNum mod 2)
        MyNum := MyNum + 1
        Mode := Get
if      Mode = Put and not Me = Sender
then    Q(You) := Q(You) ++ Msg(Null,(MyNum)mod 2)
        LastMsg := Msg(Null,(MyNum mod 2)
        MyNum := MyNum + 1
        Mode := Get
if      Mode = Get and Defined(InMsg(Me)) and
        Num(InMsg(Me))=(MyNum mod 2) and Me = Receiver
then    File(MyNum) := Data(InMsg(Me))
        Mode := Put
```

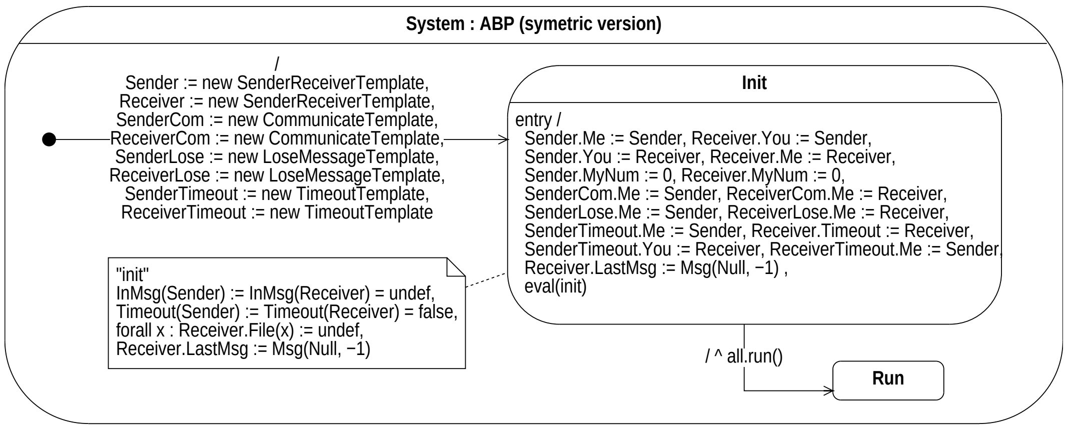

System : ABP (symetric version)
/
Sender := new SenderReceiverTemplate,
Receiver := new SenderReceiverTemplate,
SenderCom := new CommunicateTemplate,
ReceiverCom := new CommunicateTemplate,
SenderLose := new LoseMessageTemplate,
ReceiverLose := new LoseMessageTemplate,
SenderTimeout := new TimeoutTemplate,
ReceiverTimeout := new TimeoutTemplate
Init
entry /
Sender.Me := Sender, Receiver.You := Sender,
Sender.You := Receiver, Receiver.Me := Receiver,
Sender.MyNum := 0, Receiver.MyNum := 0,
SenderCom.Me := Sender, ReceiverCom.Me := Receiver,
SenderLose.Me := Sender, ReceiverLose.Me := Receiver,
SenderTimeout.Me := Sender, Receiver.Timeout := Receiver,
SenderTimeout.You := Receiver, ReceiverTimeout.Me := Sender,
Receiver.LastMsg := Msg(Null, −1) ,
eval(init)
"init"
InMsg(Sender) := InMsg(Receiver) = undef,
Timeout(Sender) := Timeout(Receiver) = false,
forall x : Receiver.File(x) := undef,
Receiver.LastMsg := Msg(Null, −1)
/ ^ all.run()
Run

Figure 15: STD for ABP System, symmetric version

```
if      Mode = Get and Defined(InMsg(Me)) and
        Num(InMsg(Me))=(MyNum mod 2) and
        not Me = Receiver
then    Mode := Put
if      Mode = Get and Defined(InMsg(Me)) and
        not Num(InMsg(Me))=(MyNum mod 2)
        or Timeout(Me)
then    Q(You) := Q(You) ++ LastMsg
        Timeout(Me) := false
if      Mode = Get and Defined(InMsg(Me))
then    Clear(InMsg)
```

Module: Communication Template

```
if      Undefined(InMsg(Me)) and not Q(Me) = EmptyQueue
then    InMsg(Me) := Head(Q(Me))
        Q(Me) := Tail(Q(Me))
```

Module: LoseMessage Template

```
if      v ≤ Q(Me)
then    Q(Me) := v
```

Module: LoseMessage Template

```
if      Undefined(InMsg(Me)) and Q(Me) = EmptyQueue and
        Undefined(InMsg(You)) and Q(You) = EmptyQueue
then    Timeout(Me) := true
```

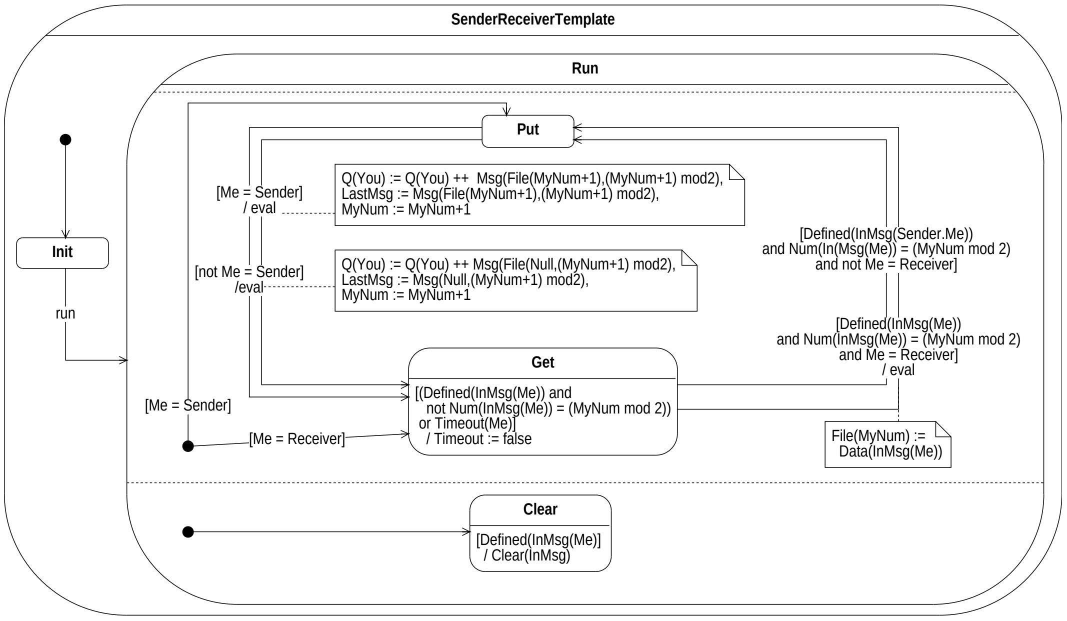

SenderReceiverTemplate
Run
Init
run
Put
[Me = Sender] / eval
Q(You) := Q(You) ++ Msg(File(MyNum+1),(MyNum+1) mod2), LastMsg := Msg(File(MyNum+1),(MyNum+1) mod2), MyNum := MyNum+1
[not Me = Sender] /eval
Q(You) := Q(You) ++ Msg(File(Null,(MyNum+1) mod2), LastMsg := Msg(Null,(MyNum+1) mod2), MyNum := MyNum+1
[Defined(InMsg(Sender.Me)) and Num(In(Msg(Me)) = (MyNum mod 2) and not Me = Receiver]
[Defined(InMsg(Me)) and Num(InMsg(Me)) = (MyNum mod 2) and Me = Receiver] / eval
Get
[(Defined(InMsg(Me)) and not Num(InMsg(Me)) = (MyNum mod 2)) or Timeout(Me)] / Timeout := false
[Me = Sender]
[Me = Receiver]
File(MyNum) := Data(InMsg(Me))
Clear
[Defined(InMsg(Me)] / Clear(InMsg)

Figure 16: STD for ABP SenderReceiver Template, symmetric version

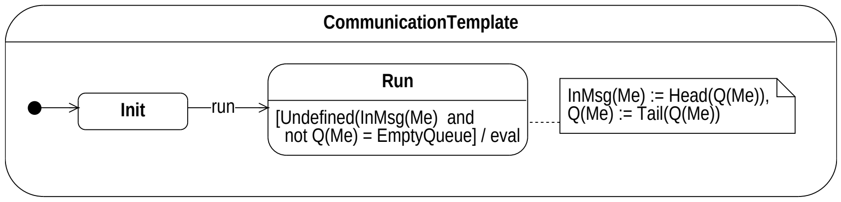

CommunicationTemplate
Init
run
Run
[Undefined(InMsg(Me) and not Q(Me) = EmptyQueue] / eval
InMsg(Me) := Head(Q(Me)), Q(Me) := Tail(Q(Me))

Figure 17: STD for ABP Communicate Template, symmetric version

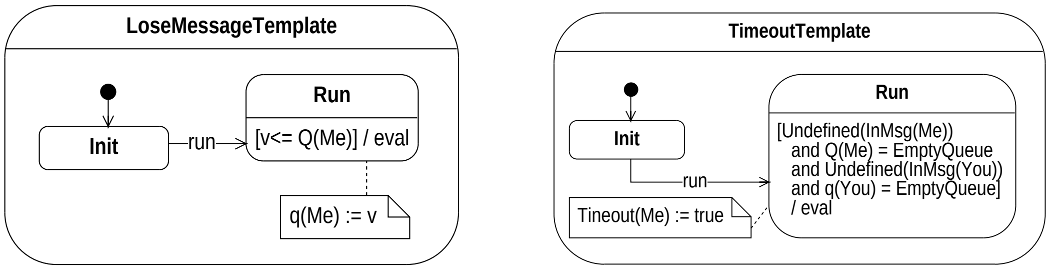

LoseMessageTemplate
Init
run
Run
[v<= Q(Me)] / eval
q(Me) := v
TimeoutTemplate
Init
run
Run
[Undefined(InMsg(Me)) and Q(Me) = EmptyQueue and Undefined(InMsg(You)) and q(You) = EmptyQueue] / eval
Tineout(Me) := true

Figure 18: STD for ABP LoseMessage, and Timeout Template, symmetric version

References

[Bör99] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and W. Stephan and P. Traverso and M. Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.

[BR94] E. Börger and D. Rosenzweig. The WAM – Definition and Compiler Correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1994.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur97] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan Department of Electrical Engineering and Computer Science, May 7, 1997. available at ftp://ftp.eecs.umich.edu/techreports/cse/1997/CSE-TR-336-97.ps.Z.

[Har87] D. Harel. Statecharts: A visual formalism for complex system. *Science of Computer Programming*, 8(3):231–274, 1987.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hug95] J. Huggins. Kermit: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 247–293. Oxford University Press, 1995.

[Obj99] Object Management Group. *OMG Unified Modelling Language Specification, Version 1.3*, June 1999. available at `http://www.rational.com/uml/index.jtmpl`.