

# The Petri Nets to Statecharts Transformation Case

Pieter Van Gorp

Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands.

`p.m.e.v.gorp@tue.nl`

Louis M. Rose

Department of Computer Science, University of York, UK.

`louis.rose@york.ac.uk`

TODO:

- Abstract
- Add a running example to the transformation section?
- Evaluation criteria
- Break down into sub-tasks. A possible ordering:
  - initialisation
  - transition precedence
  - OR rules
  - AND rules
  - Put it all together

## 1 Introduction

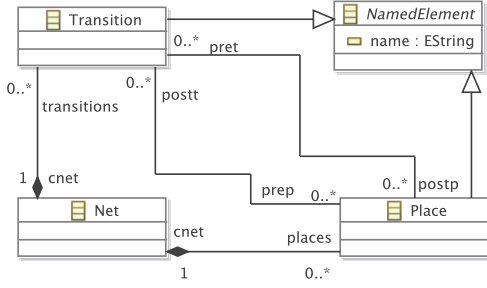
## 2 The Transformation

This section details the Petri net to state chart transformation algorithm, originally described by Eshuis [1]. The transformation described below is *input-destructive* (elements of the Petri net model are removed as the transformation proceeds), and uses the metamodels shown in figure

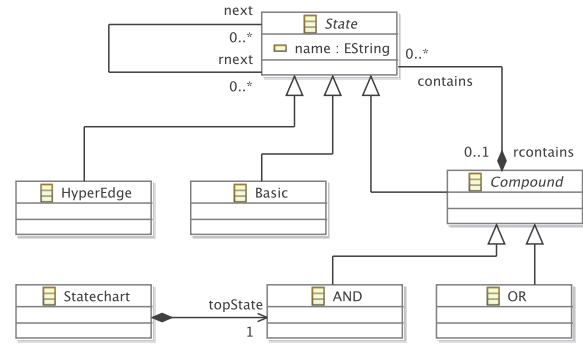
### 2.1 Initialisation

The first step in the PN2SC transformation involves creating an initial structure for the state chart model. In particular, the following state chart model elements are created:

- For every `Place` in the Petri net:
  - an instance of `Basic`,  $b$
  - an instance of `OR`,  $o$  such that  $o.contains = \{b\}$ .
- A single instance of `Statechart`,  $s$
- A single instance of `AND`,  $a$  such that:
  - $s.topstate = a$
  - $\forall o : o.contains = a$  (i.e., every instance of `OR` is contained in  $a$ )



(a) Petri net metamodel



(b) State chart metamodel

Figure 1: The metamodels used in the PN2SC transformation.

**Equivalence.** Initialisation should also provide a mechanism for identifying the OR node created for a particular Place. The precise mechanism can vary over implementations. One approach is to use a name-based identification (i.e., assign all Places a uniquely identifying name and copy each Place's name to its OR node during initialisation.) In the remainder of this section we assume that the initialisation of the transformation will construct an injective function  $equiv : Place \rightarrow OR$ .

## 2.2 Reduction rules

Following initialisation, the transformation continues by applying one of two types of reduction rule: AND and OR. This section describes these two types of reduction rule, after briefly discussing the order in which the rules should be applied.

### 2.2.1 Precedence

Correctness of the PN2SC transformation necessitates two types of precedence for the application of the AND and OR merging rules. Firstly, transitions that are more deeply nested take precedence over less nested transitions, as described below. Secondly, AND reductions always take precedence over OR reductions.

A (partial) ordering of transitions can be computed by using the ordering constraint,  $\prec$ , described by Eshuis [1]. For the transitions  $t_1$  and  $t_2$ :  $t_1 \prec t_2 \Rightarrow (t_1.prep \subset t_2.postp \vee t_1.postp \subset t_2.prep)$ , where  $\subset$  is used to mean a strict and non-empty subset. A transition  $t$  can be reduced (using the AND and OR rules described below) iff there exists no other transition  $t'$  such that  $t' \prec t$ . Figure

### 2.2.2 AND rules

The first type of reduction rule, AND (figure

**Pre-conditions.** The AND rule can be applied to a Transition,  $t$ , iff  $|t.prep| > 1$  and every Place in  $t.prep$  is connected to the same set of outgoing transitions and the same set of incoming transitions. Alternatively, the AND rule can be applied to a Transition,  $t$  iff  $|t.postp| > 1$  and every Place in  $t.postp$  is connected to the same set of outgoing transitions and the same set of incoming transitions.

```

1 operation Transition precedes?(other : Transition) : Boolean {
2   if (self == other) return false;
3
4   return self.prep.subset?(other.postp) or
5         self.postp.subset?(other.prep);
6 }
7
8 operation Collection subset?(other : Collection) : Boolean {
9   if (self.isEmpty()) return false;
10
11  // (non-strict subset and not equal) => strict subset
12  return other.includesAll(self) and not other.asSet().equals(self.asSet());
13 }

```

Figure 2: EOL code for computing the precedence of transitions.

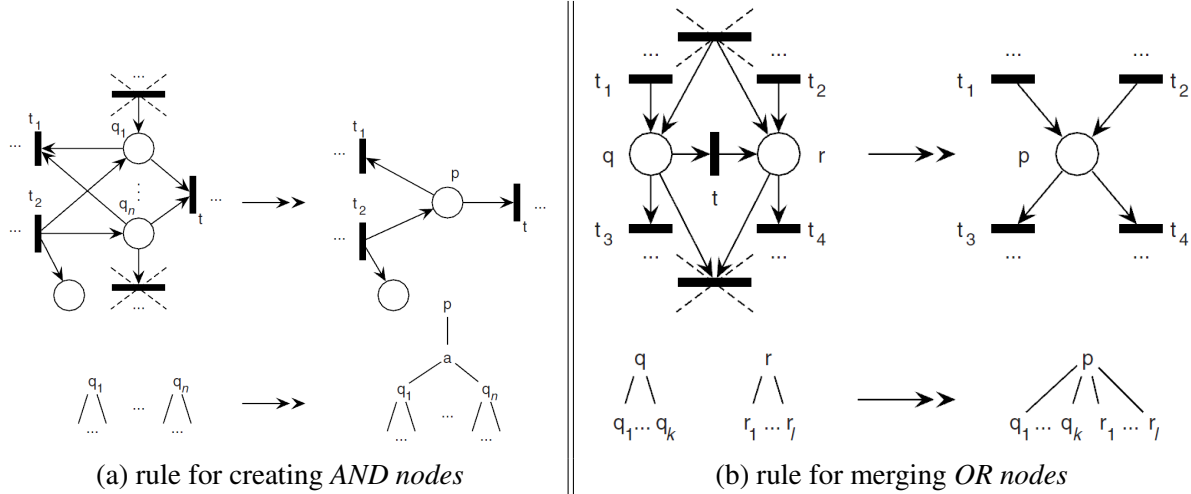


Figure 3: Visual documentation for mapping rules.

**Effect on statechart.** Applying the AND rule results in the creation of a new AND state ( $a$ ) and a new OR state ( $p$ ) such that  $p.contains = a$  and  $a.contains$  is the set of OR states  $\{q \in t.prep : equiv(q)\}$ ; or  $\{q \in t.postp : equiv(q)\}$  if the rule has been applied to the transition's postset,  $t.postp$ .

**Effect on Petri net.** Applying the AND rule removes from the Petri net all but one of the Places in the set  $t.prep$  ( $t.postp$ ).

**EOL Implementation.** Figure

### 2.2.3 OR rules

The second type of reduction rule, OR (figure

**Pre-conditions.** The OR rule can be applied to a Transition,  $t$ , iff  $(|t.prep| = 1) \wedge (|t.postp| = 1)$  and there is no transition,  $t'$ , such that  $(q \in t'.prep) \wedge (r \in t'.prep)$  or  $(q \in t'.postp) \wedge (r \in t'.postp)$  where  $q$  is the single place contained in  $t.prep$  and  $r$  is the single place contained in  $t.postp$ .

**Effect on statechart.** Applying the OR rule results in the creation of a new OR state ( $p$ ) such that  $p.contains$  is the set of OR states  $equiv(q).contains \cup equiv(r).contains$ .

```

1 operation PN!Transition apply_and_rule() : Boolean {
2   return self.apply_and_rule(self.prep) or self.apply_and_rule(self.postp);
3 }
4
5 operation PN!Transition apply_and_rule(places : Collection(PN!Place)) : Boolean {
6   var result = false;
7
8   // Check pre-conditions
9   if (places.size() > 1 and places.forAll(p|p.has_same_transitions_as?(places.first))) {
10
11     // Alter statechart
12     var parent = new SC!AND;
13     parent.contains.addAll(places.equivalentent());
14
15     var root = new SC!OR;
16     get_top_state().contains.add(root);
17     root.contains.add(parent);
18
19     // Alter Petri net
20     delete places.tail();
21
22     result = true;
23   }
24
25   return result;
26 }
27
28 operation PN!Place has_same_transitions_as?(other : PN!Place) : Boolean {
29   return self.pret.size() == other.pret.size() and
30         self.postt.size() == other.postt.size() and
31         self.pret.forAll(e|other.pret.includes(e)) and
32         self.postt.forAll(e|other.postt.includes(e));
33 }
34
35 operation Collection tail() : Collection {
36   var tailed = self.clone();
37   tailed.removeAt(0);
38   return tailed;
39 }
40
41 operation PN!Place equivalentent() : SC!OR {
42   // This operation is implementation-specific.
43   // In our solution, we use the names of Places and OR
44   // nodes to implement equivalence, but other approaches
45   // are equally valid and correct.
46 }

```

```

1 operation Transition apply_or_rule() : Boolean {
2   var result = false;
3
4   if (self.prep.size() == 1 and self.postp.size() == 1) {
5     var q = self.prep.first;
6     var r = self.postp.first;
7
8     if ((q == r) or not q.shares_a_transition_with?(r)) {
9       if (q <> r) {
10        var merger = q.equivalent();
11        var mergee = r.equivalent();
12
13        // Update statechart, re-using the q state as p
14        merger.contains.addAll(mergee.contains);
15        delete mergee;
16
17        // Update Petri net, re-using the q place as p
18        q.pret.addAll(r.pret);
19        q.postt.addAll(r.postt);
20        delete r;
21        delete self;
22
23        result = true;
24      }
25    }
26  }
27
28  return result;
29 }
30
31 operation Place shares_a_transition_with?(other : Place) : Boolean {
32   return self.pret.exists(e|e.postp.includes(other)) or
33     self.postt.exists(e|e.prep.includes(other));
34 }

```

**Effect on Petri net.** Applying the OR rule removes from the Petri net the Transition  $t$  and the Places  $q$  and  $r$ ; and adds a new Place  $p$  such that  $p.pret = (q.pret \cup r.pret)$  and  $p.postt = (q.postt \cup r.postt)$ .

**EOL Implementation.** Figure

### 2.3 Termination

The transformation proceeds by applying AND or OR rules until no Transitions remain in the Petri net (success) or until no further AND or OR rules can be applied (failure). The transformation might fail if the input Petri net is not well-formed.

## 3 Evaluation Criteria

**Acknowledgements:** The authors thank Prof. Juan de Lara for constructing the initial versions of the Petri net and state charts metamodels used in this case.

## References

- [1] R. Eshuis (2009): *Translating Safe Petri Nets to Statecharts in a Structure-Preserving Way*. In A. Cavalcanti & D.R. Dams, editors: *Proc. Formal Methods (FM), Lecture Notes in Computer Science 5850*, Springer, pp. 239–255.

- [2] D.S. Kolovos, R.F. Paige & F.A. Polack (2006): *The Epsilon Object Language (EOL)*. In A. Rensink & J. Warmer, editors: *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, *Lecture Notes in Computer Science* 4066, Springer, pp. 128–142.