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

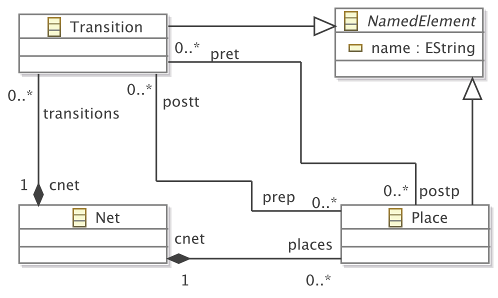
This section details the Petri net to state chart transformation algorithm, originally described by Eshuis [1]. The transformation described below is *input-destructive* (elements of the Petri net model are removed as the transformation proceeds), and uses the metamodels shown in figure 1.

2.1 Initialisation

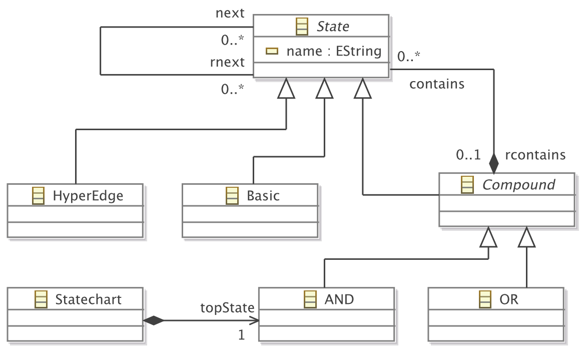
The first step in the PN2SC transformation involves creating an initial structure for the state chart model. In particular, the following state chart model elements are created:

- 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:
  $s.topstate = a$
  $\forall o : o.rcontains = a$ (i.e., every instance of `OR` is contained in a)

Submitted to:
TTC 2013



(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 eaach `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, AND reductions always take precedence over OR reductions. Secondly, transitions that are more deeply nested take precedence over less nested transitions.

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 shows an implementation of the $\prec$ in the Epsilon Object Language [2] (a reworking and extension of OCL).

2.2.2 AND rules

The first type of reduction rule, AND (figure 3.a), constructs an `AND` state for a set of `Places` that are connected to the same incoming and outgoing `Transitions`.

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.

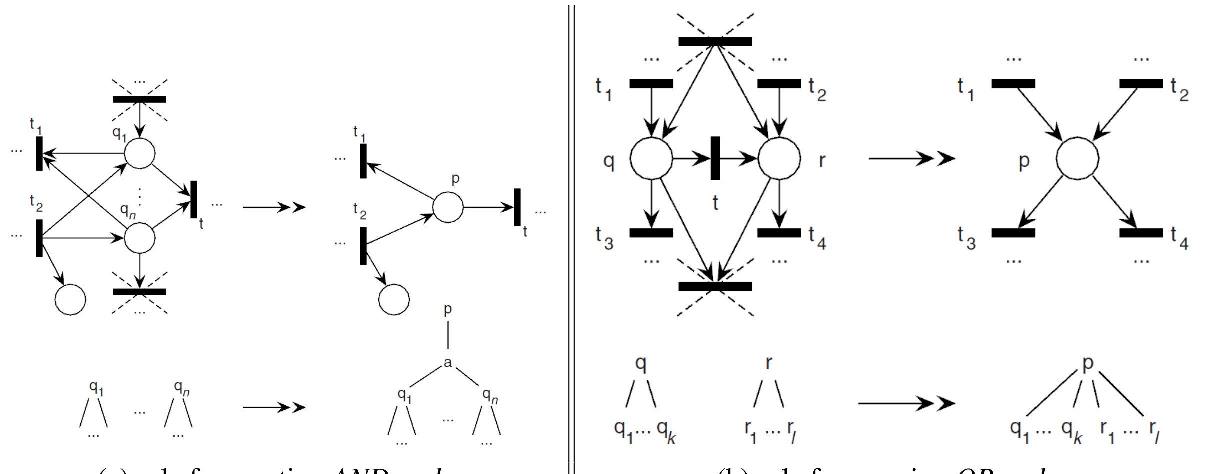
```
operation Transition precedes?(other : Transition) : Boolean {
  if (self == other) return false;

  return self.prep.subset?(other.postp) or
         self.postp.subset?(other.prep));
}

operation Collection subset?(other : Collection) : Boolean {
  if (self.isEmpty()) return false;

  // (non-strict subset and not equal) => strict subset
  return other.includesAll(self) and not other.asSet().equals(self.asSet());
}
```

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



(a) rule for creating *AND nodes* (b) rule for merging *OR nodes*

Figure 3: Visual documentation for mapping rules.

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.

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 4 shows an implementation of the AND rule in EOL. Note that the AND rule can be applied in "both directions" on line 2.

2.2.3 OR rules

The second type of reduction rule, OR (figure 3.b), constructs an `OR` state for a `Transition` that has a single preceding `Place` and single succeeding `Place`.

```
operation PN!Transition apply_and_rule() : Boolean {
  return self.apply_and_rule(self.prep) or self.apply_and_rule(self.postp);
}

operation PN!Transition apply_and_rule(places : Collection(PN!Place)) : Boolean {
  var result = false;

  // Check pre-conditions
  if (places.size() > 1 and places.forAll(p|p.has_same_transitions_as?(places.first))) {

    // Alter statechart
    var parent = new SC!AND;
    parent.contains.addAll(places.equivalent());

    var root = new SC!OR;
    get_top_state().contains.add(root);
    root.contains.add(parent);

    // Alter Petri net
    delete places.tail();

    result = true;
  }

  return result;
}

operation PN!Place has_same_transitions_as?(other : PN!Place) : Boolean {
  return self.pret.size() == other.pret.size() and
         self.postt.size() == other.postt.size() and
         self.pret.forAll(e|other.pret.includes(e)) and
         self.postt.forAll(e|other.postt.includes(e));
}

operation Collection tail() : Collection {
  var tailed = self.clone();
  tailed.removeAt(0);
  return tailed;
}

operation PN!Place equivalent() : SC!OR {
  // This operation is implementation-specific.
  // In our solution, we use the names of Places and OR
  // nodes to implement equivalence, but other approaches
  // are equally valid and correct.
}
```

Figure 4: EOL code for performing AND merges (Figure 3a).

```
operation Transition apply_or_rule() : Boolean {
  var result = false;

  if (self.prep.size() == 1 and self.postp.size() == 1) {
    var q = self.prep.first;
    var r = self.postp.first;

    if ((q == r) or not q.shares_a_transition_with?(r)) {
      if (q <> r) {
        var merger = q.equivalent();
        var mergee = r.equivalent();

        // Update statechart, re-using the q state as p
        merger.contains.addAll(mergee.contains);
        delete mergee;

        // Update Petri net, re-using the q place as p
        q.pret.addAll(r.pret);
        q.postt.addAll(r.postt);
        delete r;
        delete self;

        result = true;
      }
    }
  }

  return result;
}

operation Place shares_a_transition_with?(other : Place) : Boolean {
  return self.pret.exists(e|e.postp.includes(other)) or
         self.postt.exists(e|e.prep.includes(other));
}
```

Figure 5: EOL code for performing OR merges (Figure 3b).

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 $equiv(q).contains \cup equiv(r).contains$.

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 5 shows an implementation of the OR rule in EOL. Note that this implementation re-uses q to form p, rather than instantiate a new `Place` or `OR` state.

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.