Solving the Petri-Nets to Statecharts Transformation Case with UML-RSDS

K. Lano, S. Kolahdouz-Rahimi, K. Maroukian
Dept. of Informatics, King's College London, Strand, London, UK*

This paper provides a solution to the Petri-Nets to statecharts case using UML-RSDS. We show how a highly declarative solution which is confluent and invertible can be given using this approach.

Keywords: Petri-Nets; Statecharts; UML-RSDS.

1 Introduction

This case study [4] is an update-in-place transformation which simultaneously modifies (by deletion and simplification) an input Petri-Net model, and (by construction and elaboration) an output statechart model. We provide a specification of the transformation in the UML-RSDS language [5] and show that this is terminating, confluent and invertible.

UML-RSDS is a model-based development language and toolset, which specifies systems in a platform-independent manner, and provides automated code generation from these specifications to executable implementations (in Java, C$^{\#}$ and C++). Tools for analysis and verification are also provided. Specifications are expressed using the UML 2 standard language: class diagrams define data, use cases define the top-level services or functions of the system, and operations can be used to define detailed functionality. Expressions, constraints, pre and postconditions and invariants all use the standard OCL notation of UML 2.

For model transformations, the class diagram expresses the metamodels of the source and target models, and auxiliary data can also be defined. Use cases define the main transformation phases of the transformation: each use case has a set of pre and postconditions which define its intended functionality.

The Petri Net to statecharts transformation can be sequentially decomposed into three subtransformations: an *initialise* transformation, which copies the essential structure of the Petri Net to an initial statechart, followed by the main *pn*2*sc* reduction/elaboration transformation. A final *cleanup* transformation removes elements which do not contribute to the target structure.

Figure 1 shows the source and target metamodels of the transformation, and the three use cases representing the sub-transformations.

We extend [4] by asserting that *name* is unique for *HyperEdge*, *Basic* and *OR*:

$$HyperEdge \rightarrow isUnique(name)$$
$$Basic \rightarrow isUnique(name)$$
$$OR \rightarrow isUnique(name)$$

This means that object indexing by name can be used for these entity types: $OR[s]$ denotes the or-state with name $s : String$, for example, if such a state exists.

*Research supported by the HoRTMoDA EPSRC project

Pieter Van Gorp, Louis M. Rose, Christian Krause (Eds.):
Sixth Transformation Tool Contest (TTC 2013)
EPTCS 135, 2013, pp. 101–105, doi:10.4204/EPTCS.135.13

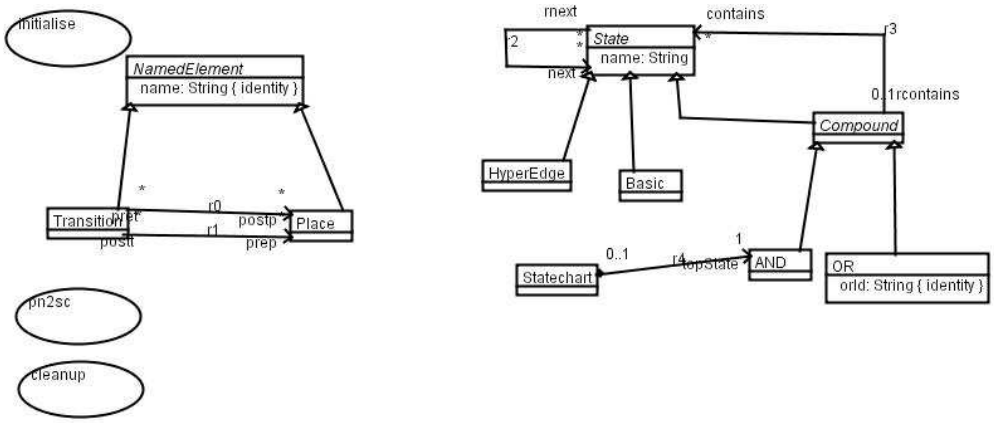

Figure 1: PN 2 SC metamodels

2 Initialisation transformation

This has the precondition that the statechart is unpopulated: $State.size = 0$, $Statechart.size = 0$, and that *name* is unique for *NamedElement*s. There are four postconditions, which define the intended state at termination of the transformation. These postconditions are also interpreted as definitions of the transformation steps.

Postcondition $I1$ applies to elements of *Place* to map them to *Basic* and *OR* states:

$Basic \rightarrow exists(\ b \mid b.name = name\ \&$
$\quad OR \rightarrow exists(\ o \mid o.name = name\ \&\ b : o.contains\ )\ )$

Logically this can be read as "for all p in *Place*, there exists b in *Basic* with $b.name = p.name$, and o in *OR* with $o.name = p.name$ and b in $o.contains$". The inverse link *rcontains* is set implicitly ($o : b.rcontains$).

Postcondition $I2$ applies to *Transition*s to map them to *HyperEdge*s:

$HyperEdge \rightarrow exists(\ e \mid e.name = name\ )$

$I3$ sets up the next/rnext links between hyperedges and basic states based upon the corresponding postt/prep links in the Petri Net:

$t : postt \Rightarrow HyperEdge[t.name] : Basic[name].next$

applied to *Place* ("if t is a post-transition of self, then the hyperedge corresponding to t is in the next states of the basic state corresponding to self").

$I4$ sets up the next/rnext links between basic states and hyperedges based upon the corresponding postp/pret links in the Petri Net:

$p : postp \Rightarrow Basic[p.name] : HyperEdge[name].next$

applied to *Transition*.

This transformation uses the 'Map objects before links' pattern [1] to separate mapping of elements and their links. It avoids the need for recursive processing: each of $I1$, ..., $I4$ can be implemented by a linear iteration over their source domains. This implementation is generated automatically by UML-RSDS as a Java program.

Termination, confluence and invertibility of such transformations follows by construction [1]. The computational complexity is linear in $NamedElement.size$. The transformation establishes $Basic \rightarrow isUnique(name)$, $HyperEdge \rightarrow isUnique(name)$ and $OR \rightarrow isUnique(name)$ because of the uniqueness of names of named elements. Indeed these properties are invariants of *initialise*.

3 Main transformation

This has as its preconditions $I1$, $I2$, $I3$, $I4$, together with the uniqueness properties of *name* for *Basic*, *HyperEdge* and *OR*, and that *AND* is empty. An invariant *Inv* asserts that for all places, there is a unique OR state with the same name:

$$Place{\rightarrow}forAll(p \mid OR{\rightarrow}exists1(o \mid o.name = p.name))$$

This ensures that there is an injective function $equiv : Place \rightarrow OR$. In our notation, $OR[p.name]$ is $equiv(p)$ for $p : Place$.

The uniqueness properties of *name* for *Basic*, *HyperEdge* and *OR* are also invariant. *Inv* is established by *initialise* because of postcondition $I1$ and the uniqueness of *name* on OR.

The highest priority rule (postcondition) is $Post1$, which performs the OR-reduction of [4] on *Transition* instances:

$$\begin{array}{l}
prep.size = 1 \ \&\ postp.size = 1 \ \& \\
q : prep \ \&\ r : postp \ \& \\
(q.pret \cap r.pret){\rightarrow}size() = 0 \ \& \\
(q.postt \cap r.postt){\rightarrow}size() = 0 \ \Rightarrow \\
\quad OR{\rightarrow}exists(\ p \mid p.name = q.name + \text{``_OR_''} + r.name \ \& \\
\qquad p.contains = OR[q.name].contains \cup OR[r.name].contains \ \& \\
\qquad q.name = p.name \ ) \ \& \\
\qquad q.pret{\rightarrow}includesAll(r.pret) \ \& \\
\qquad q.postt{\rightarrow}includesAll(r.postt) \ \& \\
\qquad r{\rightarrow}isDeleted() \ \& \\
\qquad self{\rightarrow}isDeleted()
\end{array}$$

This follows very closely the specification in [4], with $self : Transition$ playing the role of t. The updates to the Petri-Net are the last five lines, q replaces the $q \rightarrow self \rightarrow r$ structure and is renamed to match the new OR state, thus maintaining *Inv*.

For AND-reduction there are two postconditions/rules for the symmetric cases: $Post2$ merges pre-places with equivalent connectivities, and again is applied to each *Transition*:

$$\begin{array}{l}
p1 : prep \ \&\ prep.size > 1 \ \& \\
prep{\rightarrow}forAll(\ p2 \mid p1.pret = p2.pret \ \&\ p1.postt = p2.postt \ ) \ \Rightarrow \\
\quad AND{\rightarrow}exists(\ a \mid OR{\rightarrow}exists(\ p \mid \\
\qquad a : p.contains \ \&\ a.contains = OR[prep.name] \ \& \\
\qquad p.name = \text{``}AND1\_\text{''} + name \ \&\ a.name = \text{``}a1_\text{''} + name \ \& \\
\qquad p1.name = \text{``}AND1_\text{''} + name \ ) \ ) \ \& \\
\qquad (prep - \{p1\}){\rightarrow}isDeleted() \ \& \\
\qquad prep = Set\{p1\}
\end{array}$$

The last three lines define the update to the Petri-Net: all *prep* places of *self* are deleted except for $p1$, which is renamed to match the newly created OR state (therefore maintaining *Inv*).

$Post3$ merges post-places with equivalent connectivities, for each applicable *Transition*:

$$\begin{array}{l}
p1 : postp \ \&\ postp.size > 1 \ \& \\
postp{\rightarrow}forAll(\ p2 \mid p1.pret = p2.pret \ \&\ p1.postt = p2.postt \ ) \ \Rightarrow \\
\quad AND{\rightarrow}exists(\ a \mid OR{\rightarrow}exists(\ p \mid \\
\qquad a : p.contains \ \&\ a.contains = OR[postp.name] \ \&
\end{array}$$

$p.name = ``AND2\_" + name \ \& \ a.name = ``a2\_" + name \ \&$
$p1.name = ``AND2\_" + name \ ) \ ) \ \&$
$(postp - \{p1\}) \rightarrow isDeleted() \ \&$
$postp = Set\{p1\}$

This maintains *Inv* for the same reason as *Post*2.

4 Cleanup transformation

This transformation deletes OR states with empty contents:

$$contains.size = 0 \Rightarrow self \rightarrow isDeleted()$$

on *OR*.

Finally, an instance $sc : Statechart$ needs to be created, with $sc.topState$ being the unique topmost AND state produced by the main transformation, if such a state exists:

$$v = OR \rightarrow select(rcontains.size = 0) \ \& \ v.size = 1 \ \& \ ox : v \Rightarrow$$
$$AND \rightarrow exists(a \mid a.name = \text{"\_TOPSTATE\_"} \ \& \ ox : a.contains)$$

and

$$w = AND \rightarrow select(rcontains.size = 0) \ \& \ w.size = 1 \ \& \ ax : w \Rightarrow$$
$$Statechart \rightarrow exists(sc \mid sc.topState = ax)$$

This transformation is terminating and semantically correct by construction.

5 Results

Table 1 gives the test results for the performance tests for the Java 4 executable in the SHARE environment, and for the Java 6, C# and C++ executables on a standard Windows 7 laptop.

Test	*Transformation execution time*: Java 4	Java 6	C#	C++
sp200	100ms	15ms	29ms	0s
sp500	160ms	31ms	63ms	2s
sp1000	290ms	94ms	198ms	6s
sp5000	3815ms	1670ms	5069ms	161s
sp10000	13713ms	6614ms	21980ms	–
sp20000	48s	35s	87s	–
sp40000	258s	177s	468s	–
sp80000	3142s	5619s	10003s	–

Table 1: Performance test results for Java, C# and C++

The results for the Java 4, C# and Java 6 (which uses HashSet instead of Vector for sets) implementations were quite similar, which is in contrast to problems involving uni-directional associations, where the Java 6 translation is typically 100 times more efficient than the Java 4 version. C++ has efficiency

Solution Name	*Language (for all aspects)*	*Perform. optimis-ations*	5.2.1: verifi-cation	5.2.2: simu-lation	5.2.3: change-prop	5.2.4: reverse	5.2.5: debug.	5.2.6: refact-oring
UML-RSDS	UML-RSDS	E	CT	N	N	Y	N	N

Table 2: Solution table

problems for complex collection manipulations as used in this case study. All the versions may be found at `http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/pn2sc/`.

Table 2 shows the summary table completed for our solution.

The optimisation provided (for rules *Post*1, *Post*2, *Post*3) is to omit tests for the truth of the succedent of the rule (ie., the negative application condition of the rule) when applying the rule: the system can detect that a formula such as $self \rightarrow isDeleted()$ is inconsistent with the positive application condition of the rule, and therefore that there is no need to evaluate the formulae before applying the rule.

The transformation can be reversed by reversing the initialisation.

References

[1] K. Lano, S. Kolahdouz-Rahimi, *Constraint-based specification of model transformations*, Journal of Systems and Software, to appear, 2012.

[2] K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Comparison of model transformation verification approaches*, Modevva workshop, MODELS 2012.

[3] SHARE site for solution: `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC13::XP-NMF_Petri_Nets_State_C_pn2sc1.vdi`, directory Desktop/pn2scLano/pn2sc/, 2013.

[4] Pieter Van Gorp, Louis Rose, *The Petri-Nets to Statecharts Transformation Case*, Sixth Transformation Tool Contest (TTC 2013), 2013, EPTCS, this volume.

[5] UML-RSDS toolset and manual, `http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/`, 2013.