PN2SC Case Study: An EMF-IncQuery solution*

Benedek Izsó Ábel Hegedüs Gábor Bergmann Ákos Horváth
István Ráth
Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.

`{izso, hegedusa, bergmann, ahorvath, rath}@mit.bme.hu`

The paper presents a solution for the Petri-Net to Statecharts case study of the Transformation Tool Contest 2013, using EMF-IncQuery and Xtend for implementing the model transformation.

1 Introduction

Automated model transformations are a key factor in modern model-driven system engineering in order to query, derive and manipulate large, industrial models. Since such transformations are frequently integrated to modeling environments, they need to provide fast reaction time to support software engineers.

The objective of the EMF-IncQuery [3] framework is to provide a declarative way to define queries over EMF models without needing to manually code imperative model traversals. EMF-IncQuery extended the pattern language of Viatra (e.g.: with transitive closure, role navigation, match count) and tailored it to EMF models [1]. The semantics of the pattern language is similar to VTCL (published previously), but the adaptation of the rule language is an ongoing work. EMF-IncQuery uses the same incremental engine as Viatra, and latest developments extend this concept by providing a preliminary rule execution engine to perform transformations, however it is under heavy development, and the design of a dedicated rule language (instead of using the engine's API) is currently future work. The current case study aims at implementing the Petri-Net to Statecharts case study using EMF-IncQuery as a rule engine. Conceptually, this new execution environment provides a mean to specify graph transformations (GT) as rules, where the LHS (left hand side) is defined with declarative EMF-IncQuery graph patterns [1], and the RHS (right hand side) as imperative model manipulations formulated in Xtend [2]. Finally, the prototypical rule execution engine is configured from Java code, which automatically fire rules on match.

One case study of the 2013 Transformation Tool Contest describes a Petri-Net to Statecharts transformation [4]. Main characteristics of the transformation are that it i) destructs the input (Petri-Net) model during the construction of the output (Statechart) model, and ii) the transformation is divided into three phases: initialization, reduction and termination.

The rest of the paper is structured as follows: Section 2 gives an overview of the implementation, Section 3 describes the solution including design decisions, benchmark results and the solution for change propagation, and Section 4 concludes our paper.

---

*This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003), the TÁMOP (4.2.2.B-10/1–2010-0009) projects. This research was realized in the frames of TÁMOP 4.2.4. A/1-11-1-2012-0001 „National Excellence Program – Elaborating and operating an inland student and researcher personal support system". The project was subsidized by the European Union and co-financed by the European Social Fund.

Pieter Van Gorp, Louis M. Rose, Christian Krause (Eds.):
Sixth Transformation Tool Contest (TTC 2013)
EPTCS 135, 2013, pp. 106–114, doi:10.4204/EPTCS.135.14

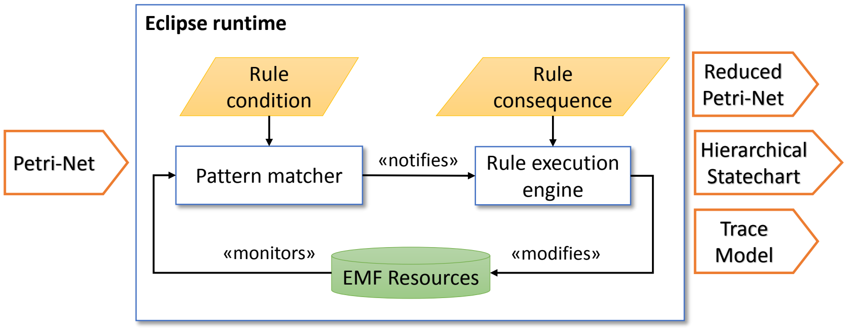


Figure 1: Overview of the specification and runtime

2 Architecture overview

The overview of the rule based solution is illustrated in Figure 1. The input of the transformation is a Petri-Net, and as a result the reduced Petri-Net, a hierarchical statechart, and an auxiliary trace model is generated. The transformation is run in the Eclipse runtime: initially it reads the input Petri-Net resource, creates the output resources (organizing them into a resource set), then executes the transformation, and finally serializes the results into files. The transformation consists of three phases: the initial mapping, the Petri-Net reduction part (applying the OR and AND rules), and the termination phase (creation of the top Statechart elements). During the process, the EMF-INCQUERY incremental pattern matcher monitors the models for satisfiable rule conditions (from the rule set of the given phase), and on match notifies the rule execution engine. Based on the specified rule consequences, the rule engine modifies models of the resource set (reduces the Petri-Net and builds the Statechart), enabling new conditions to be satisfied, thus enabling new rules to fire. While there is some satisfied precondition, the engine fires them automatically.

The whole solution is written in three languages. Rule condifions are formulated as EMF-INCQUERY graph patterns, while the rule consequences (model manipulations) in Xtend. These preconditions and rule actions are paired into rule specifications that are given to the execution engine using its Java based API.

3 Solution

3.1 Specification

The rule specification consists of two parts, which is illustrated in Figure 2. A partial solution of the AND rule demonstrates the formalization of its LHS and RHS.

The precondition of the AND rule is formulated in EMF-INCQUERY graph pattern language [1] [2], as illustrated in Figure 2a. The pattern (named *andPrecond*) can be satisfied in two ways (represented by two or-ed bodies), and returns satisfying *Place-Transition* pairs, where the place P is from the set of places from the precondition of the AND rule. The first case is described in lines 2-5, where transition T has a *pre*-place P (line 2.), *countPrePlaces* is the number of places with post-transition T (line 3), which must be at least two (expressed by a check expression in line 4.). The T post-transition must not have two

---

[1] EMF-INCQUERY language: `http://wiki.eclipse.org/EMFIncQuery/UserDocumentation/QueryLanguage`

[2] More examples and demos: `http://incquery.net/incquery/examples`

```
1  pattern andPrecond(P:Place, T:Transition) {
2    Transition.prep(T, P);
3    countPrePlaces == count find postT(_PX, T);
4    check(countPrePlaces >= 2);
5    neg find nonCommonTPost(T);
6  } or {
7    Transition.postp(T, P);
8    countPostPlaces == count find preT(_PX, T);
9    check(countPostPlaces >= 2);
10   neg find nonCommonTPre(T);
11 }
12 pattern nonCommonTPost(T:Transition) {
13   find transitionWithTwoPrePlaces(T, P1, P2);
14   find postT(P1, T1);
15   neg find postT(P2, T1);
16 } or {
17   find transitionWithTwoPrePlaces(T, P1, P2);
18   find preT(P1, T1);
19   neg find preT(P2, T1);
20 }
21 pattern postT(P, T) {Place.postt(P, T);}
22 pattern preT(P, T) {Place.pret(P, T);}
```

(a) AND rule condition (EMF-INCQUERY)

```
1  val IMatchProcessor<AndPrecondMatch> andProcessor = [
2    var EList<Place> placesSet
3    if (p.postt.contains(t)) placesSet = t.prep
4    else placesSet = t.postp
5    val newP = stf.createOR()
6    val newA = stf.createAND()
7    newA.moveTo(newP.contains)
8    newP.moveTo(stateChartResource.contents)
9    placesSet.forEach[ p |
10     equiv(p).moveTo(newA.contains)
11   ]
12   placesSet.forEach[ removeTrace ]
13   createTrace(place, newP)
14   val placeSetIt = new ArrayList(placesSet)
15   placeSetIt.forEach[if (it != place) deletePlace]
16 ]
```

(b) AND rule processor (Xtend)

Figure 2: Definition of the AND rule

pre-places with different pre- or post-transitions which is expressed by a negative application condition in line 5. The negatively called pattern finds two pre-places of *T* (lines 13,17), and in the first case (lines 14-15) checks for a post-transition of *P1* which is not a post-transition of *P2*, while in the second case (lines 18-19) it checks for a pre-transition of P1 which is not a pre-transition of P2. The second case of the AND precondition (when the transition has at least two *post*-places) can be formulated similarly. The whole code of the AND precondition and postcondition is described in Appendix A.1.

The effect of the rule is achieved by executing imperative model editing commands, formulated in Xtend. Such model manipulations build up a processor, as illustrated in Figure 2b for the AND rule. In lines 2-4 the set of places (*placeSet*) is determined by checking whether the place is a pre-place of the transition, or a post-place. In lines 5-8 the new *OR* and *AND* states are created, connected, and put into the statechart model. Then mapped places (*equiv(p)*) are moved below the newly created AND (lines 9-11). The place from the set of places selected by the pattern is reused, so after deleting old traces, a new trace is created for it, and other places are deleted from the Petri-Net (lines 12-15).

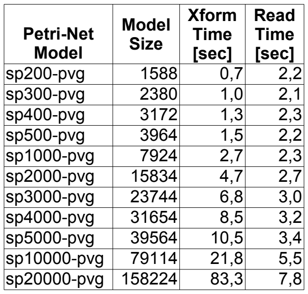
The specification of the AND rule binds the pattern *andPrecond* as LHS, and *andProcessor* as RHS using the Java API. These rules are executed automatically by the engine on match.

3.2 Benchmark results

The transformations were run on SHARE, on an Ubuntu 12.04, i686 architecture inside a VirtualBox. The CPU is an Intel Quad CPU Q9650 clocked at 3.00GHz, but in the virtualized environment only one is visible to the OS. The virtual computer has 1 GB of RAM, and 512 MB of swap space.

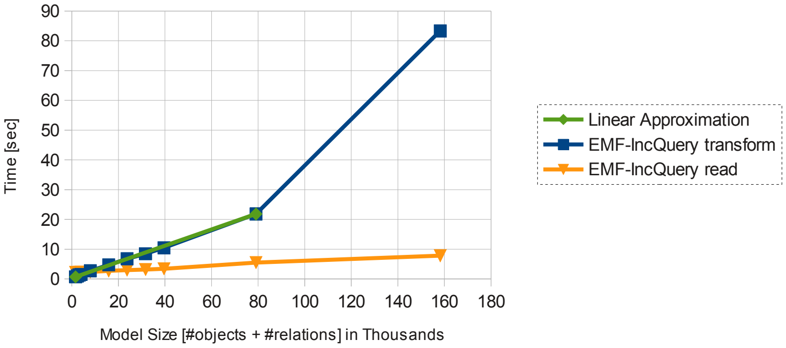
Results are displayed on Figure 3. Figure 3a shows the numerical results in tabular form, where the first column is the name of the provided benchmark model, the second is the EMF model size, the third is the transformation time in seconds, and the fourth is the read time in seconds. The model size is the sum of all objects and relations of the EMF model.

Figure 3b displays the transformation time and model size on a scatterplot. It shows that EMF-INCQUERY scaled linearly up to 80 thousand elements (sp10000-pvg) (transforming the model in 22 secs), and ran for the model containing 158 thousand elements (sp20000-pvg). As the pattern matcher is a memory-intensive application, for the largest model more than 1 GB was necessary, which involved active swapping. This degraded runtime performance (obviously because hard disk is slower than RAM), and also because the CPU intensive kswapd and the transformation program shared the same CPU. Read times were not negligible, but were orders of magnitude less than the transformation time. On one of our machines 10 GBs of memory could be given to the JVM, where transformations were run for models of all sizes. Here, the whole transformation for the sp20000-pvg model (largest model transformed on SHARE) was executed twice as fast as on SHARE. This effect can be probably attributed to less GC call, because for the smaller models, runtimes were in orders of magnitude the same. Transforming the largest model on our machine took 87 minutes, however giving (and allocating) 15 GBs of memory instead of 10GBs, speed up the same transformation to 78 minutes.



Petri-Net Model	Model Size	Xform Time [sec]	Read Time [sec]
sp200-pvg	1588	0,7	2,2
sp300-pvg	2380	1,0	2,1
sp400-pvg	3172	1,3	2,3
sp500-pvg	3964	1,5	2,2
sp1000-pvg	7924	2,7	2,3
sp2000-pvg	15834	4,7	2,7
sp3000-pvg	23744	6,8	3,0
sp4000-pvg	31654	8,5	3,2
sp5000-pvg	39564	10,5	3,4
sp10000-pvg	79114	21,8	5,5
sp20000-pvg	158224	83,3	7,8

(a) Benchmark results for EMF-IncQuery on SHARE (tabular)



(b) Benchmark results for EMF-IncQuery on SHARE (scatterplot)

Figure 3: PN2SC benchmark results on SHARE for EMF-IncQuery

These measured values are in accordance with the results published in the case study. The performance is linear for medium models, and exponential for large models, similarly to the GrGen.NET results. The hard (slow) parts for the EMF-INCQUERY tool was that this use case is model manipulation intensive, resulting in many intermediate changes of the result set of the patterns.

3.3 Optimizations

No test case specific optimizations were made, but for the whole system some special settings and best practices were applied. *Finding common subpatterns* and extracting them into a pattern results in better performance (as the engine must process only once this part), and better maintainability (instead of copy-paste code). Such named pattern in Appendix A.1 is the `tranWithTwoPostPlaces` describing a structure that can be used in both (or-ed) bodies of `nonCommonTPre`. Named patterns can be called negatively (e.g. `postT`), and can be used as preconditions (e.g.: `andPrecond`).

3.4 Transformation correctness and reproducibility

The transformation runs correctly for the provided test cases on SHARE[3], and the source code is also available on Github[4]. Automatic correctness validation was not implemented, but comparing the two models in the EMF tree editor shows equivalent structure. The transformation stops when multiple top level elements remain at the end, and creates a root element when only one top level element remains, enabling to inspect Statecharts with the provided GMF editors.

---

[3] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_EIQ-PN2SC.vdi

[4] https://github.com/izsob/TTC13-PN2SC-EIQ

3.5 Change propagation

As EMF-INCQUERY is an incremental technology, change propagation could be solved easily, using the same rule-based methodology. To handle the change of place, transition elements, or relations between them, patterns for precondition can be specified (matching only places, etc.), as illustrated in A.2.1.

Three rules are created to handle addition, deletion or name update of places and transitions with the processors described in lines 31-181 of Appendix A.2.2. The actual Petri-Net changes propagated to the target model are in lines 1-26 in A.2.2.

This can be tested by running the ”PN2SC_CP” test case on SHARE from the runtime Eclipse. This performs changes on the transformed testcase1-in.petrinet. Snapshots of the changed Petri-Net and its propagations are saved in instances/snapshots, which can be inspected using the EMF tree editor.

3.6 Tool support for debugging and refactoring

As the transformation is written in three languages, debugging and refactoring is dependent on these languages, and on engine capabilities. Firings of the transformation can be debugged by placing breakpoints in the Xtend code, and debug messages of the execution engine can be turned on, which prints useful messages about rule firings and activations. Xtend and the EMF-INCQUERY pattern editors are based on Xtext, and while refactoring capabilities exist, they are sometimes limited. Debugging declarative EMF-INCQUERY graph patterns are impossible at runtime, but when snapshots are made from the model, the snapshot (EMF model) and queries can be loaded into the Query Explorer view, which is very handy to debug matches at a given point. The engine controller code can be debugged and refactored well, as it is written in Java.

4 Conclusion

In this paper we have presented our EMF-INCQUERY based implementation for the Petri-Nets to Statechart case study. This is one of the the first cases where the prototipical execution engine based on EMF-INCQUERY is used as a rule engine, however, currently it has no dedicated rule language, and the engine is under heavy development.

The transformation is specified using declarative graph pattern queries over EMF models for rule preconditions, and Xtend code which can be executed to obtain the desired effect of the rule. Relying on incremental query evaluation of EMF-INCQUERY, the change propagations are also implemented.

References

[1] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF models*. In: *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Lecture Notes in Computer Science* 6707, Springer, pp. 167–182, doi:10.1007/978-3-642-21732-6_12.

[2] Eclipse.org: *Xtend - Modernized Java*. `http://www.eclipse.org/xtend/`.

[3] Eclipse.org (2013): *EMF-IncQuery*. `http://eclipse.org/incquery/`.

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

A Appendix - PN2SC transformation code

A.1 AND precondition as EMF-IncQuery graph patterns

The following code snippet shows the precondition of the AND rule with all dependent (called) patterns. Note that naming subpatterns (even simple ones) enhances performance, and these can be called (also negatively), or can be preconditions of rules.

```
// AND precondition
pattern andPrecond(P:Place, T:Transition) {
  // T is the transition with at least 2 pre-places
  Transition.prep(T, P);
  countPrePlaces == count find postT(_PX, T);
  check(countPrePlaces >= 2);
  neg find nonCommonTPost(T);
} or {
  // T is the transition with at least 2 post places
  Transition.postp(T, P);
  countPostPlaces == count find preT(_PX, T);
  check(countPostPlaces >= 2);
  neg find nonCommonTPre(T);
}

// T is a post-transition of P
pattern postT(P, T) {
  Place.postt(P, T);
}

// T is a pre-transition of P
pattern preT(P, T) {
  Place.pret(P, T);
}

// place without common pre or post transition,
// when T has >= 2 post places
pattern nonCommonTPre(T:Transition) {
  // T is a pre transition of P1 and P2,
  // but P1 has another pre-transition (T1), which is not a pre-transition of P2
  find tranWithTwoPostPlaces(T, P1, P2);
  find preT(P1, T1);
  neg find preT(P2, T1);
} or {
  // T is a pre transition of P1 and P2,
  // but P1 has another post-transition (T1), which is not a post-transition of P2
  find tranWithTwoPostPlaces(T, P1, P2);
  find postT(P1, T1);
  neg find postT(P2, T1);
}

// T is a transition with P1 and P2 post-places
pattern tranWithTwoPostPlaces(T:Transition, P1:Place, P2:Place) {
  find preT(P1, T);
  find preT(P2, T);
  P1 != P2;
}

// place without common pre or post transition,
// when T has >= 2 pre places
pattern nonCommonTPost(T:Transition) {
  // T is a post transition of P1 and P2,
  // but P1 has another post-transition (T1), which is not a post-transition of P2
  find transitionWithTwoPrePlaces(T, P1, P2);
  find postT(P1, T1);
  neg find postT(P2, T1);
} or {
  // T is a post transition of P1 and P2,
  // but P1 has another pre-transition (T1), which is not a pre-transition of P2
  find transitionWithTwoPrePlaces(T, P1, P2);
  find preT(P1, T1);
  neg find preT(P2, T1);
}

// T is a transition with P1 and P2 pre-places
pattern transitionWithTwoPrePlaces(T:Transition, P1:Place, P2:Place) {
  find postT(P1, T);
  find postT(P2, T);
  P1 != P2;
}
```

The following Xtend code runs on the firing of the AND rule.

```
/*
 * Rule specification of (both) "and" rule
 */
def createAndRuleSpecification() {
  val IMatchProcessor<AndPrecondMatch> processor = [
    // collect places (pre places if the transition is a post transition, post places otherwise)
    var EList<Place> placesSet
    if (p.postt.contains(t))
      placesSet = t.prep
    else
      placesSet = t.postp

    // run the AND transformation
    processAndRule(p, placesSet)
  ]

  newSimpleMatcherRuleSpecification(AndPrecondMatcher::factory,
    DefaultActivationLifeCycle::DEFAULT_NO_UPDATE_AND_DISAPPEAR,
    newHashSet(newStatelessJob(ActivationState::APPEARED, processor)))
}

/*
 * Action (processor) for the and rule
 */
def processAndRule(Place place, EList<Place> placesSet) {
    // add new OR and AND to the StateChart
    val newP = stf.createOR()
    val newA = stf.createAND()
    newA.moveTo(newP.contains)
    newP.moveTo(stateChartResource.contents)

    // add children of AND (equiv(p)) to the new AND state (newA)
    placesSet.forEach[ p | equiv(p).moveTo(newA.contains) ]

    // remove traces of places from TraceModel
    placesSet.forEach[ removeTrace ]

    // add new place --> OR (newP) to TraceModel
    createTrace(place, newP)

    // remove places from PetriNet, except one
    val placeSetIt = new ArrayList(placesSet)
    placeSetIt.forEach[
      if (it != place) deletePlace
    ]
}
```

A.2 Change propagation code

A.2.1 Precondition patterns for the change-propagation task

```
// Match places
pattern place(p) { Place(p); }

// Match transitions
pattern transition(t) { Transition(t); }

// T is a post-transition of P
pattern postT(P, T) { Place.postt(P, T); }

// T is a pre-transition of P
pattern preT(P, T) { Place.pret(P, T); }
```

A.2.2 Source model manipulation and target model modification functions in Xtend

```
def manipulate() {
  // create petrinet factors and get root place
  val onlyPlace = petriNet.places.head;

  //  a)
  //  create place and add to petrinet
  val place = pnf.createPlace();
  place.name = "newPlace";
  petriNet.places += place;
  //  create transition and add to petrinet
  val transition = pnf.createTransition();
  transition.name = "newTransition";
  transition.moveTo(petriNet.transitions)
  //  connect: place -> transition
```

```
  place.postt += transition
  //  connect: transition -> onlyPlace
  transition.postp += onlyPlace

  //  b) change names
  place.name = "theNewPlace";
  transition.name = "theNewTransition";

  //  c) remove place and transition
  deletePlace(place);
  deleteTransition(transition);
}

/*
 * Change propagation of a place
 */
def createCPPlaceRule() {
  // new place appeared
  val IMatchProcessor<PlaceMatch> processorAdd = [
    // create new basic state, and trace between place and basic
    val basic = stf.createBasic()
    basic.name = p.name
    basic.moveTo(stateChartResource.contents)
    createTrace(p, basic)

    doAllSnapshot("NewPlace")
  ]

  // a place deleted
  val IMatchProcessor<PlaceMatch> processorDelete = [
    val place = it.p

    // lookup trace of place and delete the basic
    val basic = equiv(place)
    stateChartResource.contents.remove(basic)
    removeTrace(place)

    doAllSnapshot("DeletePlace")
  ]

  // a place's name updated
  val IMatchProcessor<PlaceMatch> processorUpdate = [
    // lookup trace of place and update the basic's name
    val basic = equiv(p)
    basic.name = p.name

    doAllSnapshot("UpdatePlace")
  ]

  newSimpleMatcherRuleSpecification(PlaceMatcher::factory,
    DefaultActivationLifeCycle::DEFAULT,
    newHashSet( EnableableJob::newEnableableJob(ActivationState::APPEARED, processorAdd),
          EnableableJob::newEnableableJob(ActivationState::DISAPPEARED, processorDelete),
          EnableableJob::newEnableableJob(ActivationState::UPDATED, processorUpdate)
      ))
}

/*
 * Change propagation of a transition
 */
def createCPTransitionRule() {
  // a transition is added
  val IMatchProcessor<TransitionMatch> processorAdd = [
    val hyperEdge = stf.createHyperEdge()
    hyperEdge.name = t.name
    hyperEdge.moveTo(stateChartResource.contents)
    createTrace(t, hyperEdge)

    doAllSnapshot("NewTransition")
  ]

  // a transition is deleted
  val IMatchProcessor<TransitionMatch> processorDelete = [
    val hyperEdge = equiv(t)
    stateChartResource.contents.remove(hyperEdge)
    removeTrace(t)

    doAllSnapshot("DeleteTransition")
  ]

  // a transition's name is updated
  val IMatchProcessor<TransitionMatch> processorUpdate = [
    val hyperEdge = equiv(t)
    hyperEdge.name = t.name

    doAllSnapshot("UpdateTransition")
  ]

```

```
  newSimpleMatcherRuleSpecification(TransitionMatcher::factory,
    DefaultActivationLifeCycle::DEFAULT,
    newHashSet( EnableableJob::newEnableableJob(ActivationState::APPEARED, processorAdd),
          EnableableJob::newEnableableJob(ActivationState::DISAPPEARED, processorDelete),
          EnableableJob::newEnableableJob(ActivationState::UPDATED, processorUpdate)
    ))

}

/*
 * Change propagation of a place --> transition connection
 */
def createCPPlaceToTransitionRule() {
  // a P->T connection is created
  val IMatchProcessor<PostTMatch> processorAdd = [
    val basic = equiv(p)
    val hyperEdge = equiv(t)

    basic.next += hyperEdge

    doAllSnapshot("CP_PT_added")
  ]

    // a P->T connection is deleted
    val IMatchProcessor<PostTMatch> processorRemove = [
    val basic = equiv(p)
    val hyperEdge = equiv(t)

    basic.next.remove(hyperEdge)

    doAllSnapshot("CP_PT_removed")
  ]

  newSimpleMatcherRuleSpecification(PostTMatcher::factory,
    DefaultActivationLifeCycle::DEFAULT_NO_UPDATE,
    newHashSet(newStatelessJob(ActivationState::APPEARED, processorAdd),
             newStatelessJob(ActivationState::DISAPPEARED, processorRemove)
    ))
}

/*
 * Change propagation of a transition --> place connection
 */
def createCPTransitionToPlaceRule() {
  // a T->P connection is created
  val IMatchProcessor<PreTMatch> processorAdd = [
    val basic = equiv(p)
    val hyperEdge = equiv(t)

    hyperEdge.next += basic

    doAllSnapshot("CP_TP_added")
  ]

  // a T->P connection is deleted
  val IMatchProcessor<PreTMatch> processorRemove = [
    val basic = equiv(p)
    val hyperEdge = equiv(t)

    hyperEdge.next.remove(basic)

    doAllSnapshot("CP_TP_removed")
  ]

  newSimpleMatcherRuleSpecification(PreTMatcher::factory,
    DefaultActivationLifeCycle::DEFAULT_NO_UPDATE,
    newHashSet(newStatelessJob(ActivationState::APPEARED, processorAdd),
             newStatelessJob(ActivationState::DISAPPEARED, processorRemove)
    ))
}

def getCPRules() {
  newHashSet(
    createCPPlaceRule() as RuleSpecification<? extends IPatternMatch>,
    createCPTransitionRule() as RuleSpecification<? extends IPatternMatch>,
    createCPPlaceToTransitionRule() as RuleSpecification<? extends IPatternMatch>,
    createCPTransitionToPlaceRule() as RuleSpecification<? extends IPatternMatch>
  )
}
```