EMF Models Transformations with Tom

UNIVERSITÉ DE LORRAINE · Inria

jeanchristophe.bach@inria.fr,
pierre-etienne.moreau@loria.fr and marc.pantel@enseeiht.fr

INP ENSEEIHT · QUARTEFT · IRIT · Loria Laboratoire lorrain de recherche en informatique et ses applications · cnrs

Context

- Importance of MDE in software engineering: models transformations are an essential part
- Two main approaches to write models transformations:
  - using a General Purpose language (GPL) such as Java+EMF
  - using a DSL such as ATL, Kermeta, QVT, *etc.*
- Our approach: embedding a dedicated transformation language in a GPL by using Tom
- Need of safe software for critical systems: checking all steps of the development chain is mandatory. Therefore it is important to write qualified models transformations. Perspectives: use of the generated *links model* for verification and specification purpose.

Tom language

Tom is a language which extends general purpose languages (C, C#, Caml, Java, Python, *etc.*) by addings features:

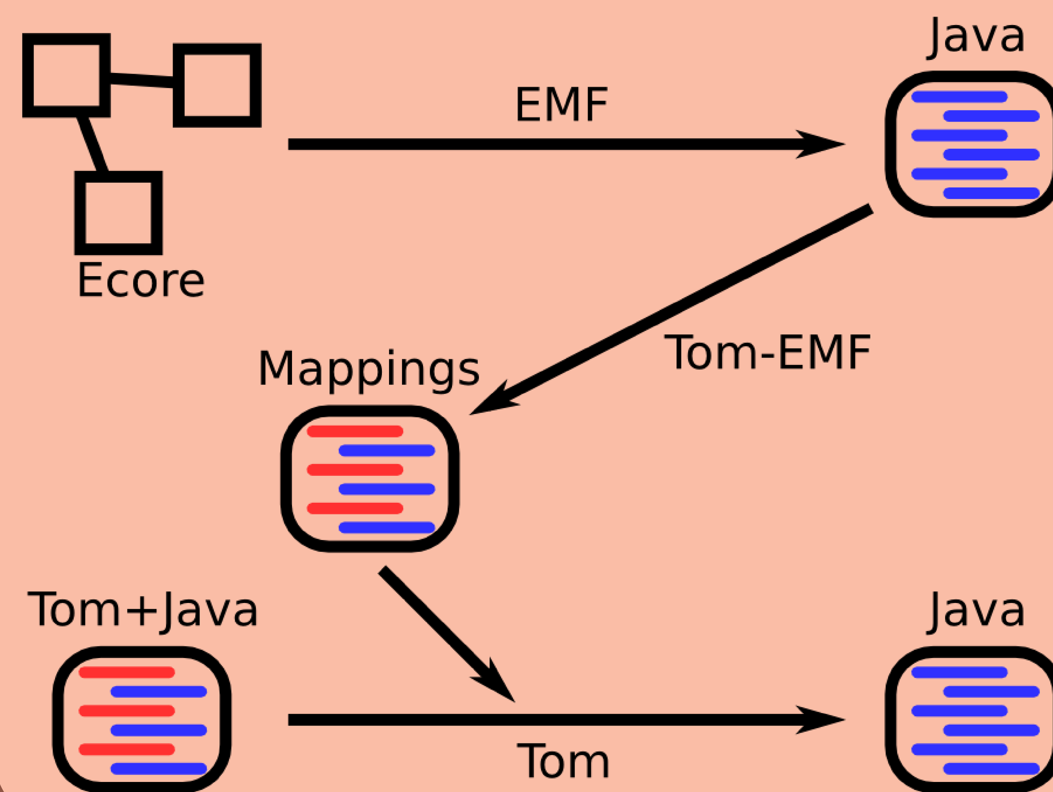
- pattern-matching
- rewriting
- strategies
- mappings (representation of external data structures as algebraic terms)

Tom code is written inside the Java program and a Tom block may contain Java code which may also contain Tom code. The Tom compiler compiles Tom constructs without parsing host code, and dissolves them into the host language.
Tom language implements the principle of *Formal Islands* as following:

```
public class Test {
  %typeterm Person {
    implement {  Person  }
    is_sort(t) {  t instanceof Person  }
  }
  %op Person person(firstname:String,
lastname:String) {
    is_fsym(t) {  t instanceof Person  }
    get_slot(firstname,t) {  t.getFirstname()  }
    get_slot(lastname,t) {  t.getLastname()  }
    make(t0, t1) {  new Person(t0, t1)  }
  }
  public static void main(String[] args) {
    Person subject =  `Person("John","Doe") ;
    %match(subject) {
     Person(first,"Doe")-> {  System.out.println(
"There is someone from Doe family:"+ `first ); }
    ...
    }
  }
}
```

http://tom.loria.fr

Tom-EMF: manipulating EMF models with Tom



Tom-EMF is a mappings generator: it takes a Java-EMF metamodel as input and generates mappings which allow to see an EMF model as a Tom term (a tree).

In addition, by using a dedicated tool called `EcoreIntrospector`, Tom strategies can be used to traverse and to rewrite the model.

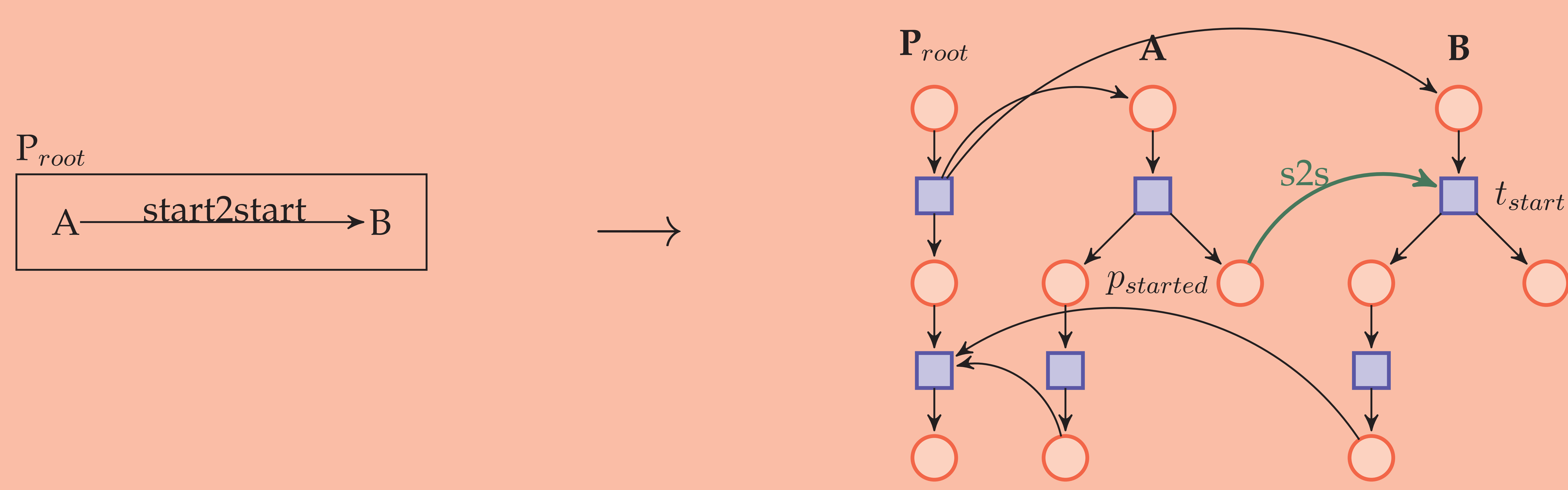
Extension of Tom

We propose to extend the Tom language to be able to write easily models transformations such as `SimplePDLToPetriNet` one by adding an higher-level construct: **%transformation**, composed of elementary transformations whose **order is not important**.
Other constructs such as **%tracelink** and **%resolve** allow us to maintain links between source and target elements during the transformation. The generated *links model* is used by a *generated links resolution strategy* at the end of the strategy. By applying constraints on it (*e.g.* OCL), it is also useful to verify a transformation.

Example of transformation code:

```
%transformation SimplePDLToPetriNet() :
  simplepdl.ecore -> petrinet.ecore {
    definition P2PN traversal `TopDown(P2PN()) {
      p@Process[name=n, from=f] -> {
        ...
        %tracelink(t_start:Transition,
                   `Transition[name=`n]);
        ...
      }
    }
    definition WD2PN traversal `BottomUp(WD2PN()) {
      wd@WorkDefinition[name=n] -> {
        ...
        Process p = `wd.getParent();
        Transition source =
           %resolve(p:WorkDefinition,
                    t_start:Transition);
        ...
      }
    }
    definition WS2PN traversal `TopDown(WS2PN()) {
      ws@WorkSequence -> { ... }
    }
  }
```

Use case: SimplePDLToPetriNet transformation



- Decomposition of the transformation into three elementary transformations: `ProcessToPetriNet`, `WorkDefinitionToPetriNet` and `WorkSequenceToPetriNet`
- Addition of intermediate elements to represent elements which may not have already been created
- Application of a `Resolve` strategy after all transformation steps