Modellbasierte Softwareentwicklung (MODSOFT)

Part II

Domain Specific Languages

Semantics

Prof. Joachim Fischer /
Dr. Markus Scheidgen / Dipl.-Inf. Andreas Blunk

{fischer,scheidge,blunk}@informatik.hu-berlin.de
LFE Systemanalyse, III.310

Agenda

prolog (1 VL) — **Introduction:** languages and their aspects, modeling vs. programming, meta-modeling and the 4 layer model

0. (2 VL) — **Eclipse/Plug-ins:** eclipse, plug-in model and plug-in description, features, *p2*-repositories, *RCP*s

1. (2 VL) — **Structure:** *Ecore*, *genmodel*, working with generated code, constraints with *Java* and *OCL*, *XML/XMI*

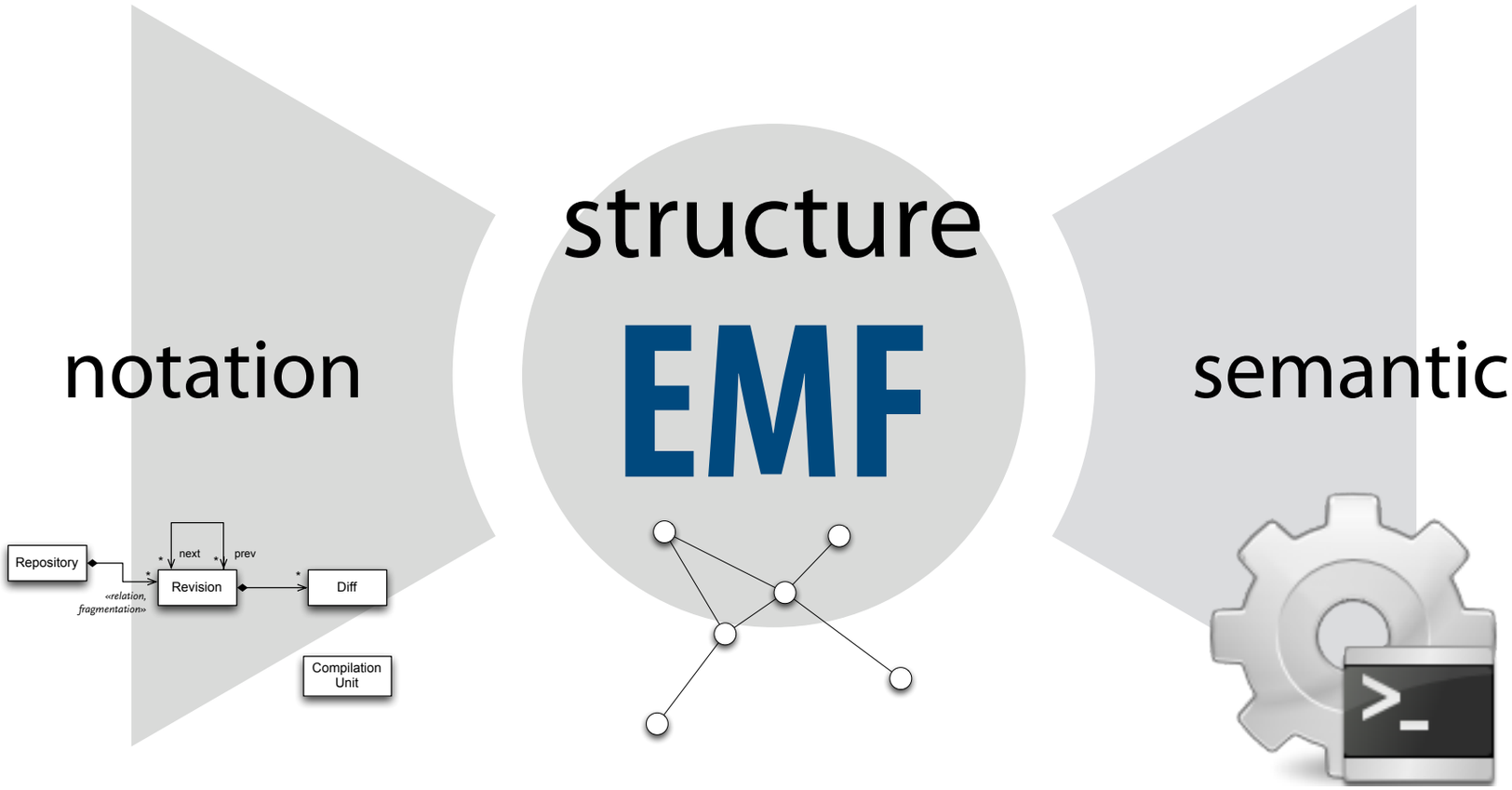
2. (3 VL) — **Notation:** Customizing the tree-editor, textural with *XText*, graphical with *GEF* and *GMF*

➡ 3. (4 VL) — **Semantics:** interpreters with Java, code-generation with *Java* and *XTend*, model-transformations with *Java* and *ATL*

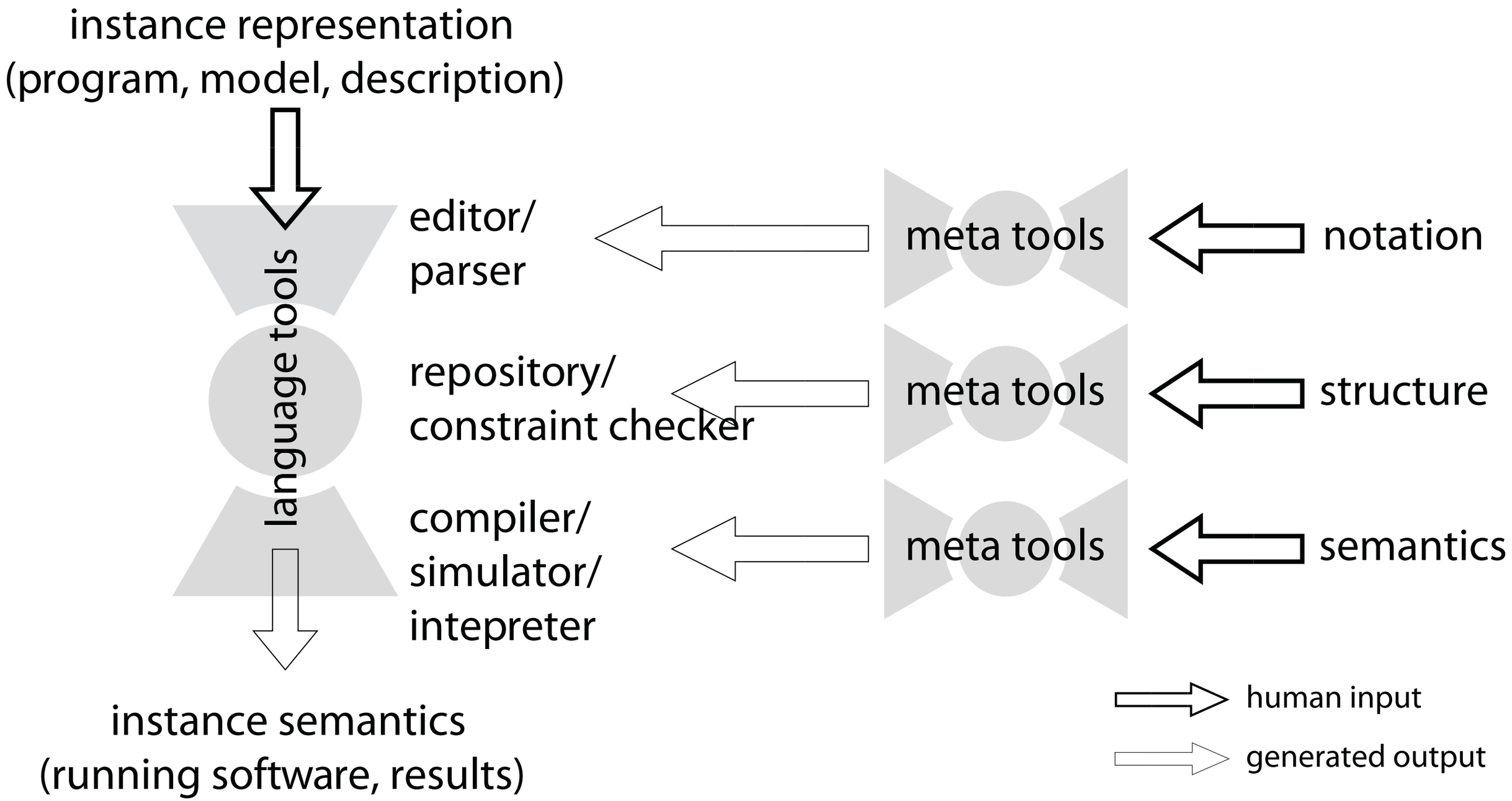
epilog (2 VL) — **Tools:** persisting large models, model versioning and comparison, model evolution and co-adaption, modular languages with *XBase*, *Meta Programming System* (MPS)

Previously on MODSOFT

Eclipse Modeling Framework



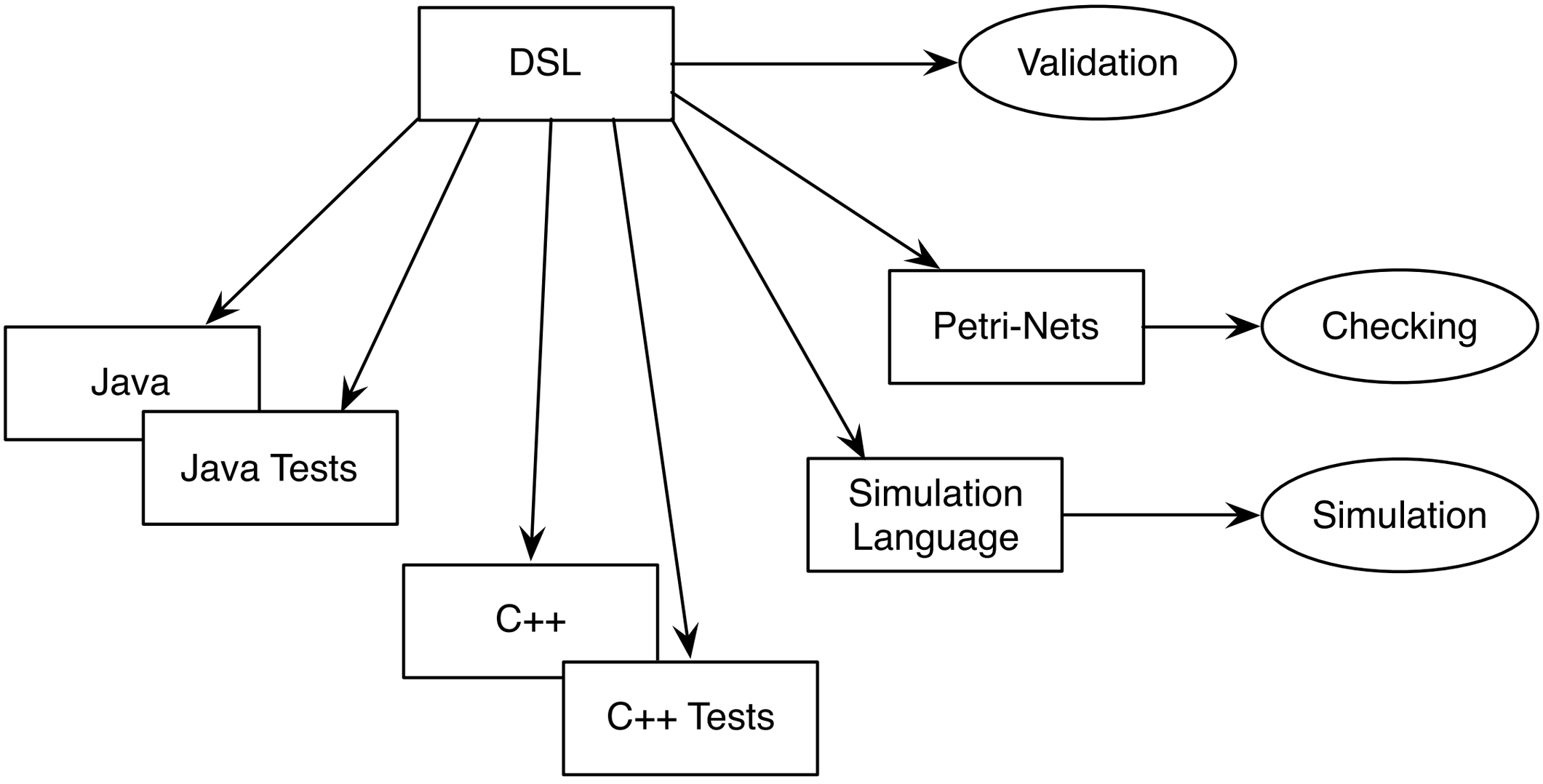
Meta-Languages



Different Types of Semantics

- **Operational Semantics**
- Denotational Semantics
- Axiomatic Semantics
- **Translational Semantics**

Semantics and DSLs



Operational Semantics

with EMF

Approaches to Operational Semantics with EMF

- Programming
  - Java or other JRE compatible languages (e.g. Groovy, Scala)
  - other languages via XMI/XML
- Action languages
  - imperative description of meta-model method implementations
    - e.g. UML Activities and UML Action Language
- Graph rewriting
  - declarative description of execution steps
  - semantics as a series of in-place model-transformations
  - like term rewriting on context-free syntax (terms), but on EMF-models (graphs)

Abstract Syntax and Runtime Concepts

- Abstract syntax covers all concepts that can be used to write models/programs before the model/program is executed
- Runtime concepts are necessary to model program/model state while the model/program is executed
- Runtime concepts can be realized within or outside of EMF
- Runtime concepts are often instances of syntax concepts
  - remember Multi-Level-Meta-Modeling with *ambiguous instantiation* and *replication of concepts*
- Runtime concepts can also be implemented in an existing runtime library, without any EMF or model connections

Meta-Model Operations and Operational Semantics

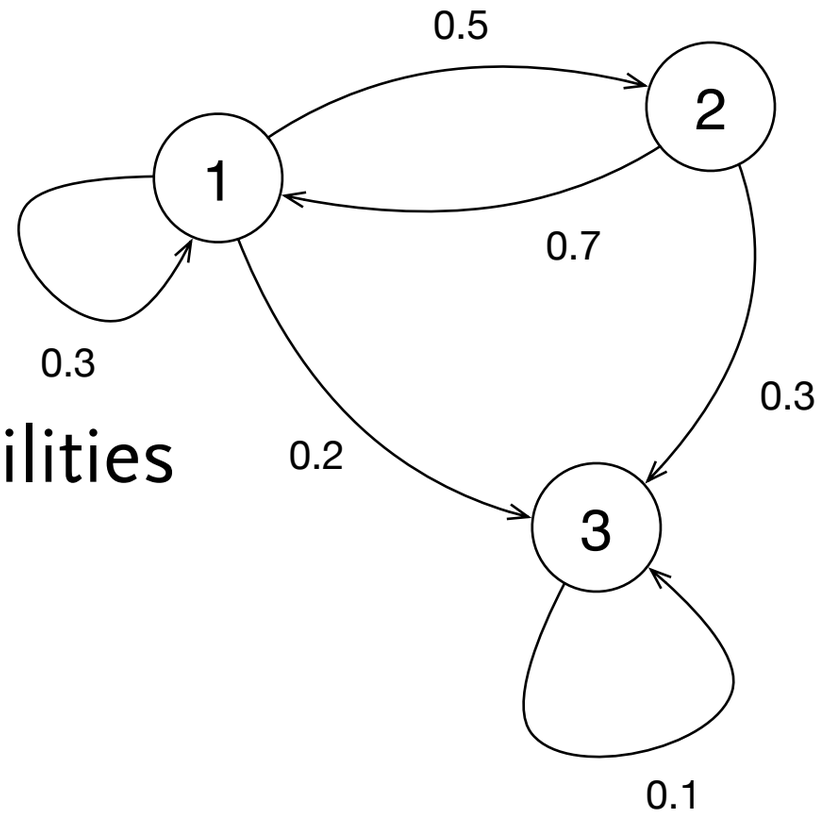
- EMF classes can declare operations
- Main operation to start interpretation
- Model as a start configuration of objects
- operation implementations can create and destroy model object
- syntax becomes runtime state

Implementation of EMF operations

- Java
- delegation to external implementations in other languages
- e.g. action languages
- e.g. *Actions*
  - UML activities to choreograph actions on the model
  - Actions are
    - instantiation
    - modification of value sets
    - destruction of objects
    - call operations
  - OCL can be used to describe expressions to compute decisions, values, and operation arguments
  - Actions can be reversed

Java Example for Operational Semantics

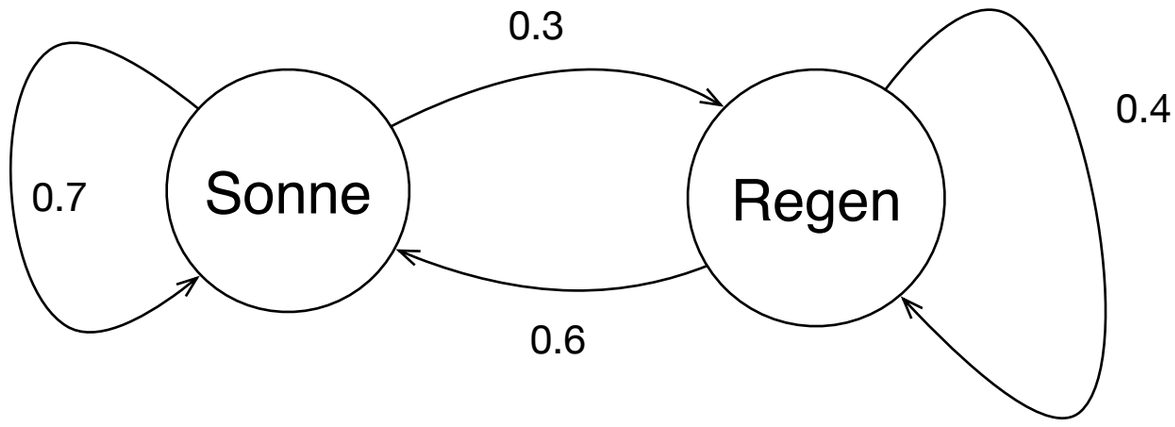
- Markov chains
  - finite number of labeled states
  - directed edges with constant probabilities
  - sum of all outgoing probabilities = 1



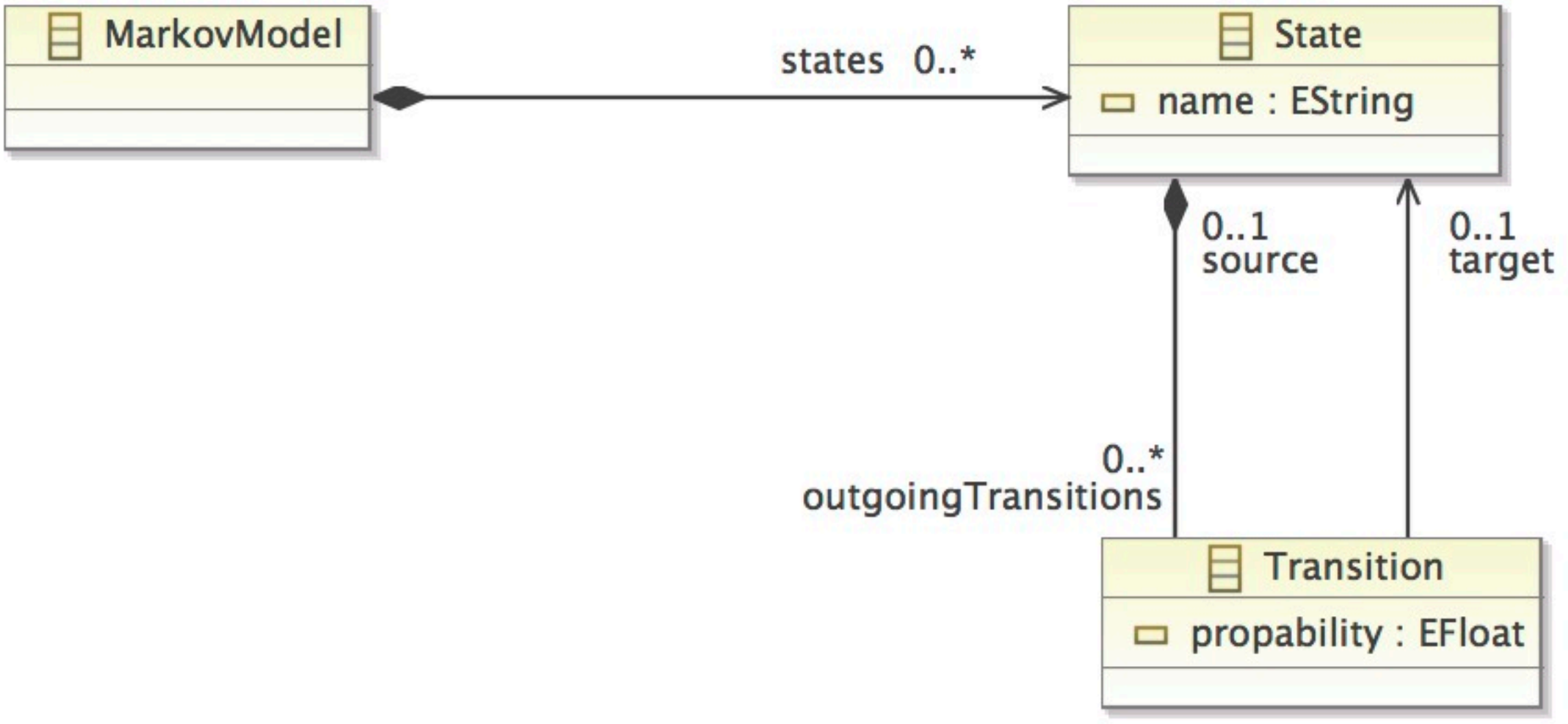
$$M \cdot s_n = s_{n+1}$$

$$\begin{pmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.7 \\ 0.3 \end{pmatrix}$$

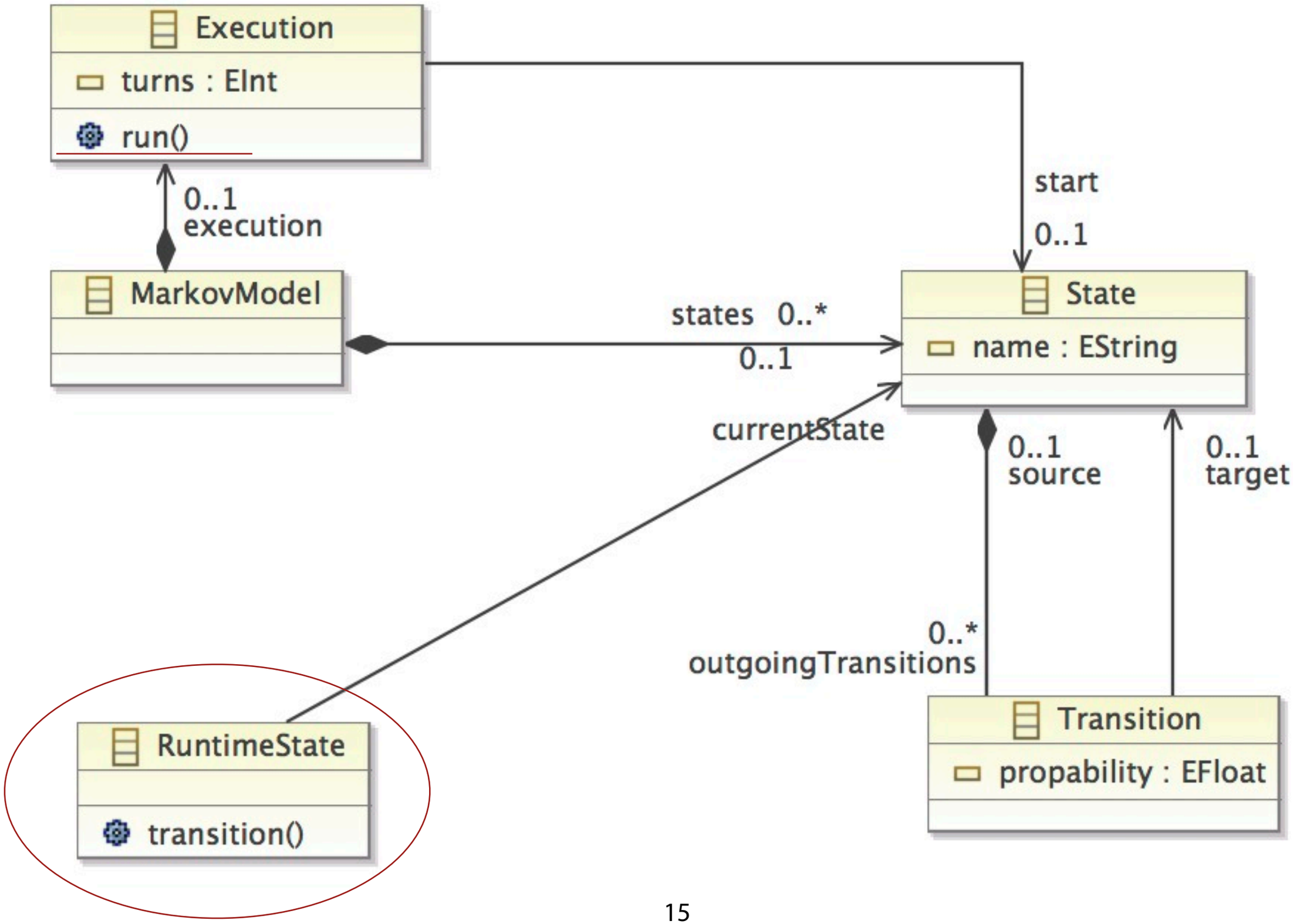
$$\begin{pmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{pmatrix} \cdot \begin{pmatrix} 0.7 \\ 0.3 \end{pmatrix} = \begin{pmatrix} 0.67 \\ 0.33 \end{pmatrix}$$



Java Example for Operational Semantics



Java Example for Operational Semantics

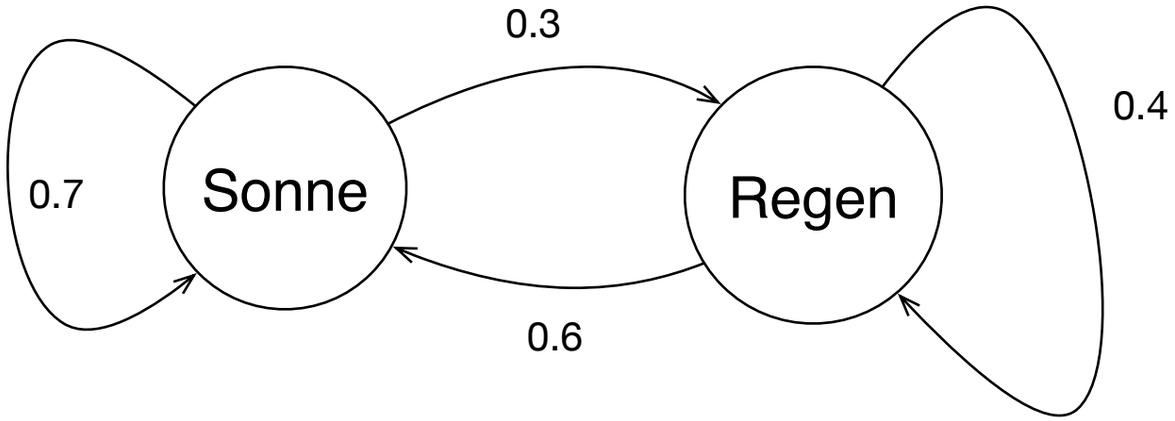


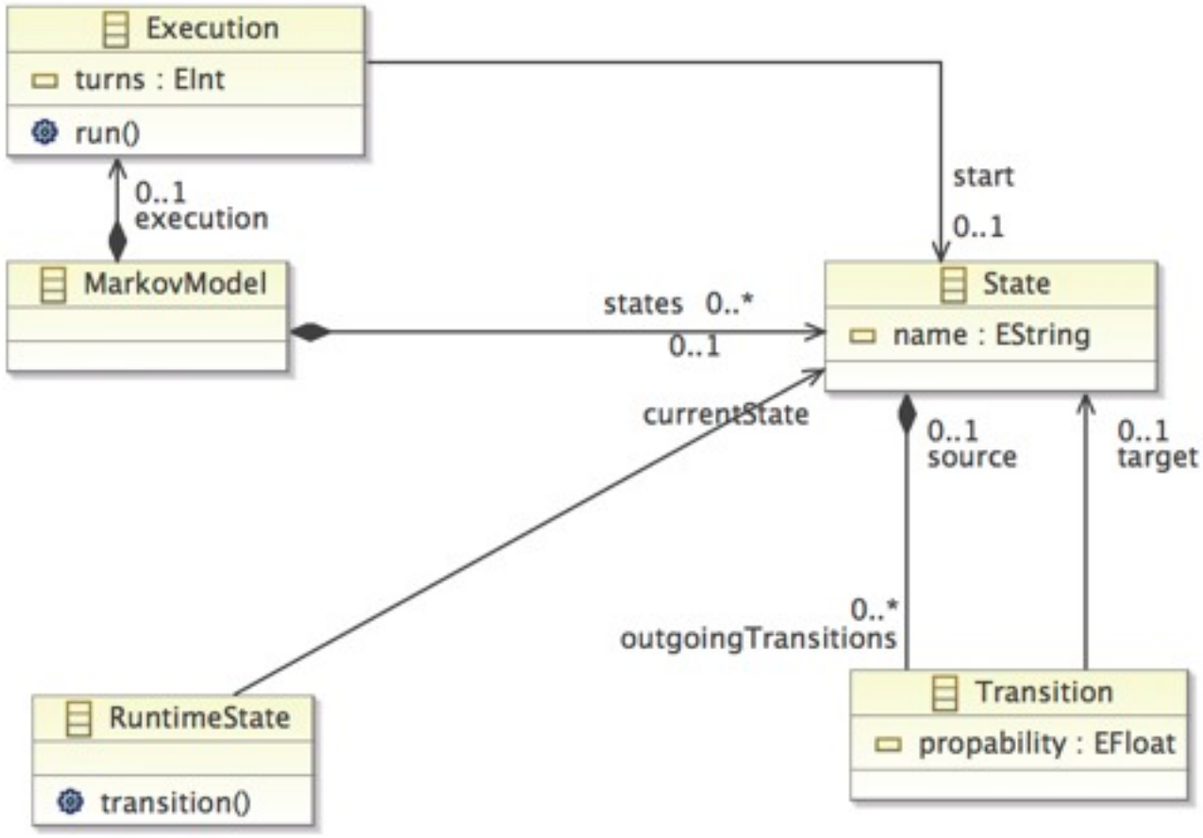
Java Example for Operational Semantics

```
public void transition() {
    double random = Math.random();
    double propabilitySum = 0;
    for(Transition transition: getCurrentState().getOutgoingTransitions()) {
        propabilitySum += transition.getPropability();
        if (random <= propabilitySum) {
            setCurrentState(transition.getTarget());
            return;
        }
    }

    throw new IllegalArgumentException("Sum of all transition propabilities must by 1");
}
```

```
public void run() {
    RuntimeState rs = MarkovFactory.eINSTANCE.createRuntimeState();
    rs.setCurrentState(getStart());
    for (int i = 0; i < getTurns(); i++) {
        System.out.println(rs.getCurrentState());
        rs.transition();
    }
}
```

Java Example for Operational Semantics

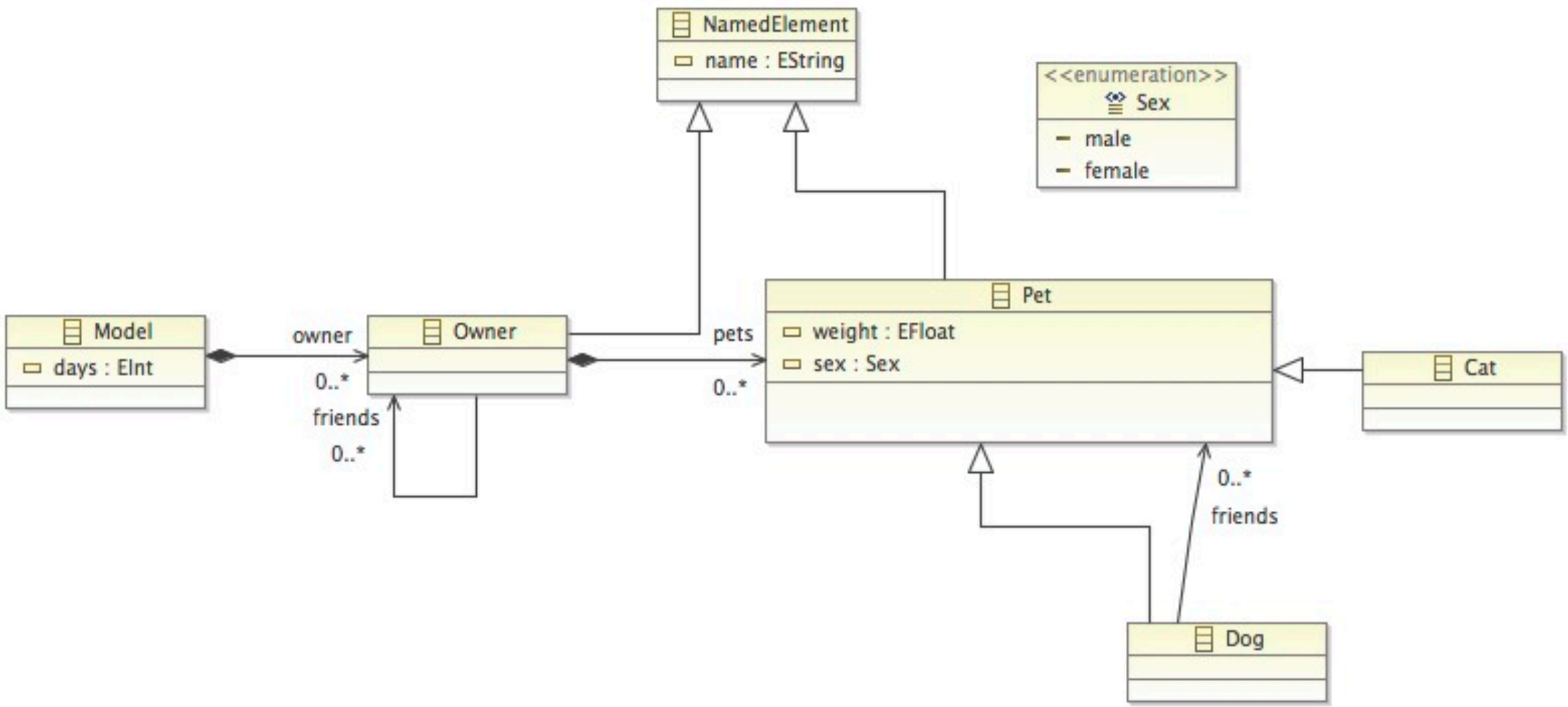




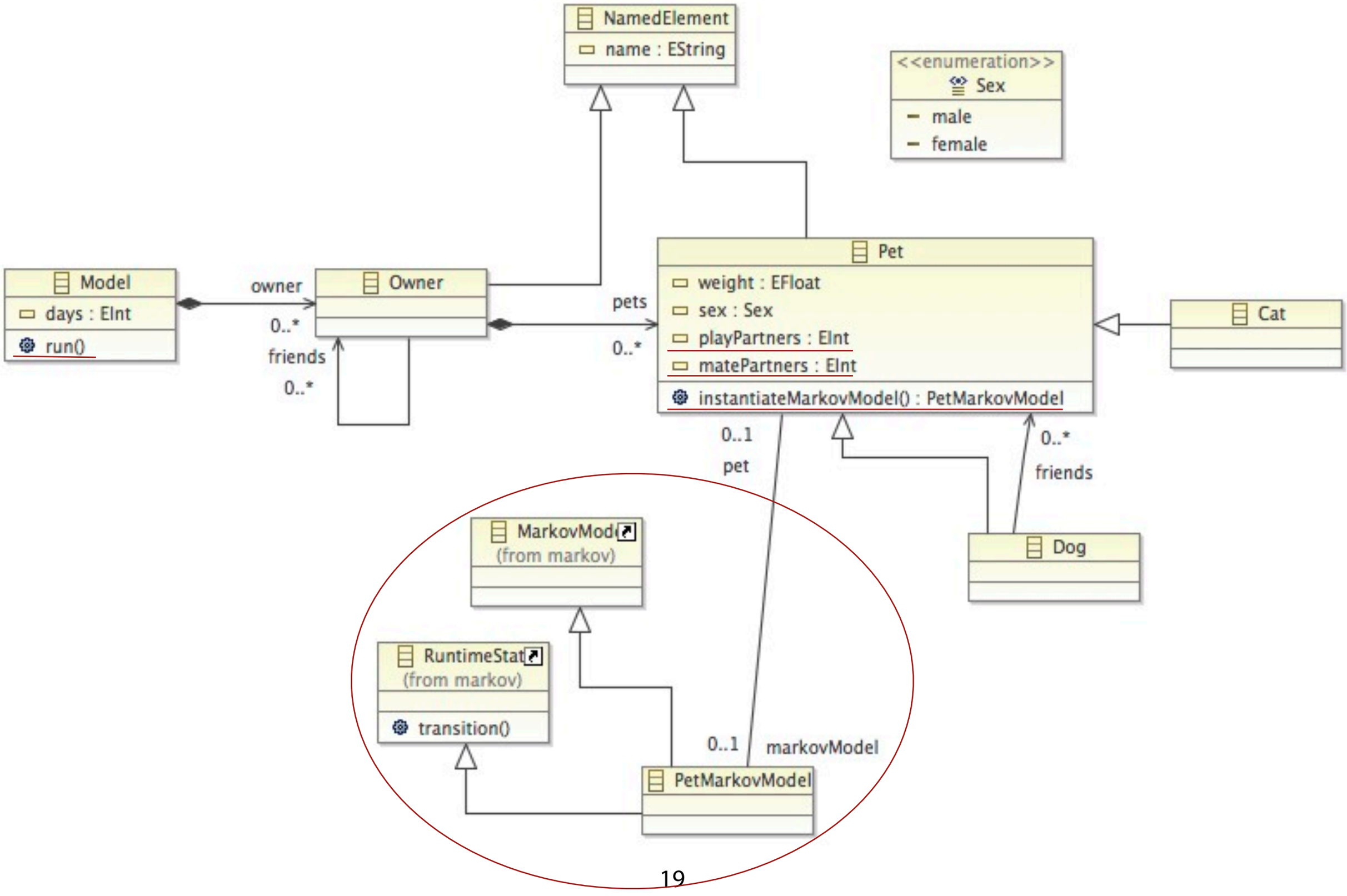

```
Sonne
Regen
Sonne
Sonne
Regen
```

```
run from sonne for 5

sonne
 to regen with 0.3
 to sonne with 0.7

regen
 to sonne with 0.6
 to regen with 0.4
```

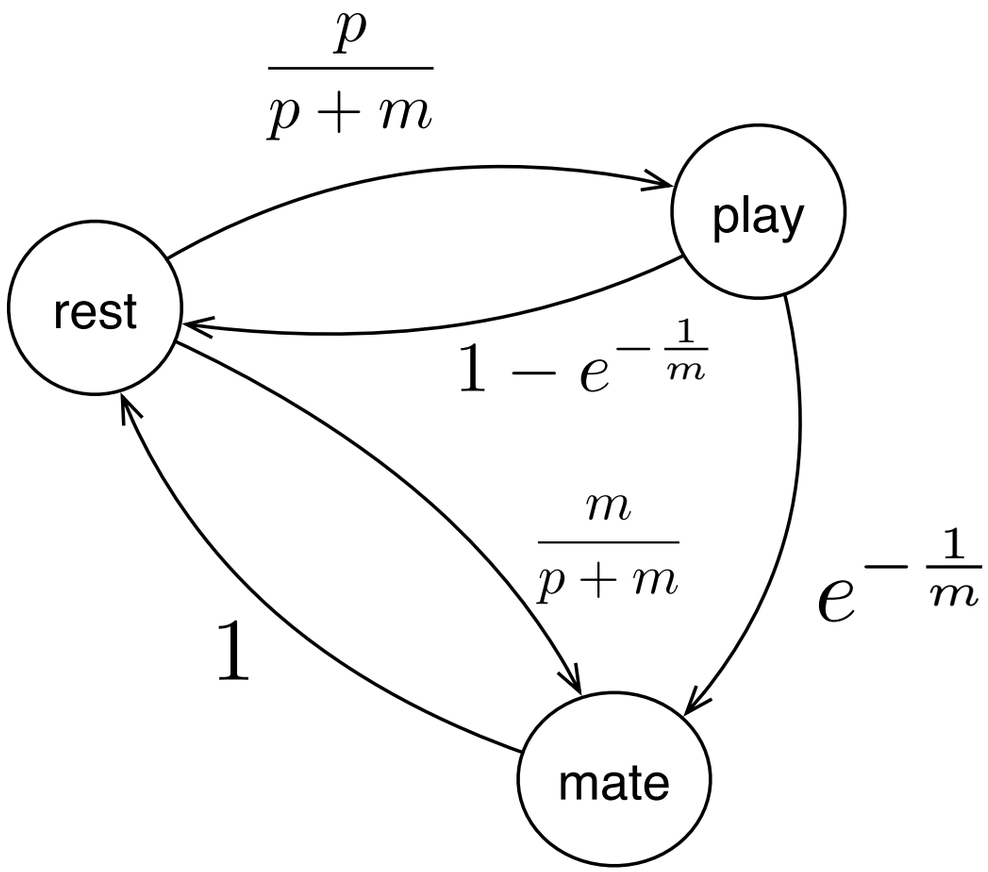
Another Example



Another Example

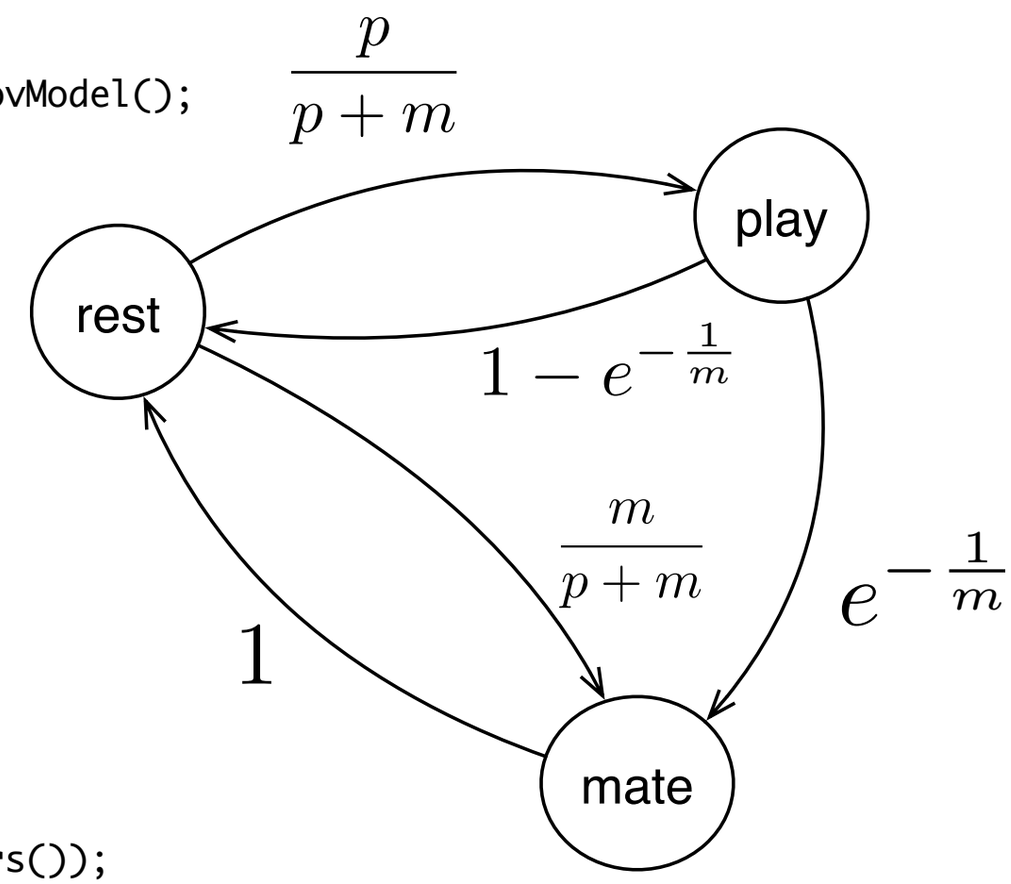


Another Example



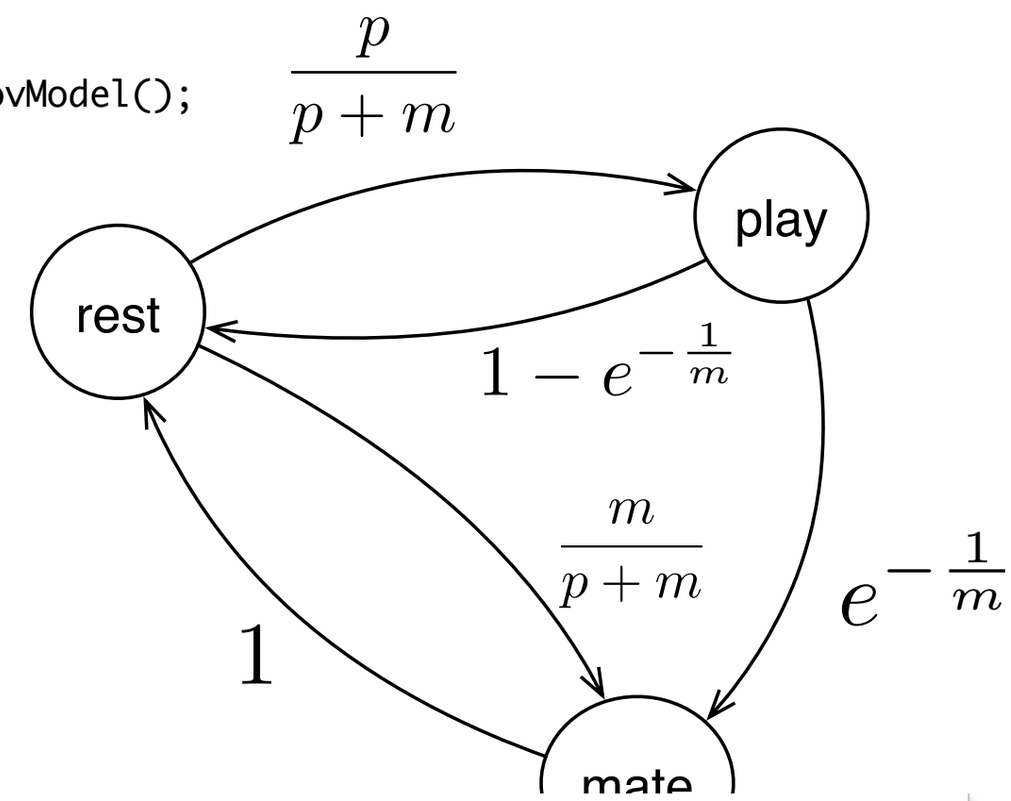
Another Example

```
public PetMarkovModel instantiateMarkovModel() {
    PetMarkovModel markovModel = FidoFactory.eINSTANCE.createPetMarkovModel();

    State play = createState("play", markovModel);
    State mate = createState("mate", markovModel);
    State rest = createState("rest", markovModel);

    markovModel.setCurrentState(rest);

    float partners = getPlayPartners() + getMatePartners();
    if (partners == 0) {
        createTransition(rest, rest, 1);
    } else {
        createTransition(rest, play, getPlayPartners()/partners);
        createTransition(rest, mate, getMatePartners()/partners);
    }

    double matePropability = Math.pow(Math.E, 1/(float)getMatePartners());
    createTransition(play, mate, (float)matePropability);
    createTransition(play, mate, (float)(1-matePropability));

    createTransition(mate, rest, 1);

    setMarkovModel(markovModel);
    return markovModel;
}
```



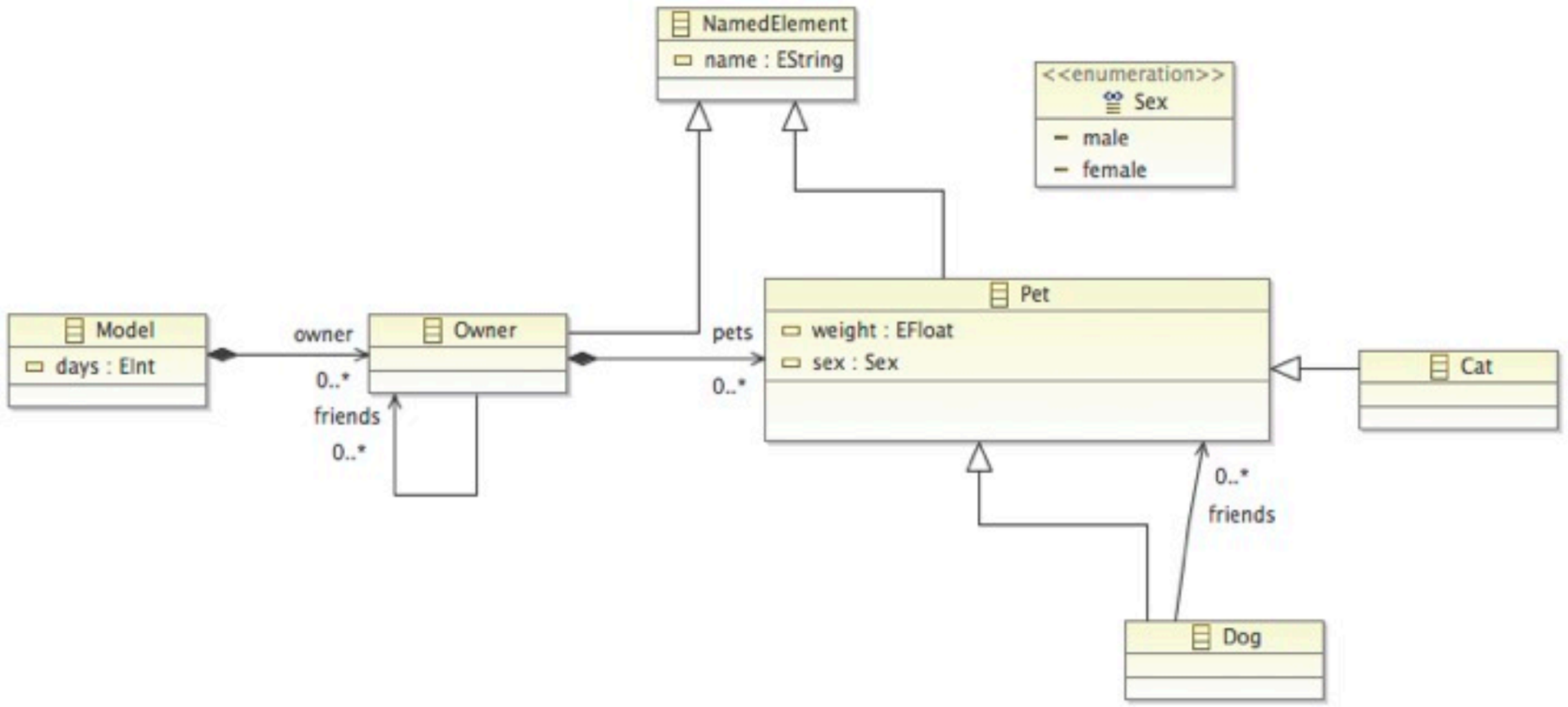
Another Example

```
public PetMarkovModel instantiateMarkovModel() {
    PetMarkovModel markovModel = FidoFactory.eINSTANCE.createPetMarkovModel();

    State play = createState("play", markovModel);
    State mate = createState("mate", markovModel);
    State rest = createState("rest", markovModel);

    markovModel.setCurrentState(rest);

    float partners = getPlayPartners() + getMatePartners();
    if (partners == 0) {
        createTransition(rest, rest, 1);
    } else {
        createTransition(rest, play, getPlayPartners()/partners);
        createTransition(rest, mate, getMatePartners()/partners);
    }

    double matePropability = M
    createTransition(play, mat
    createTransition(play, mat

    createTransition(mate, res

    setMarkovModel(markovModel
    return markovModel;
}
```

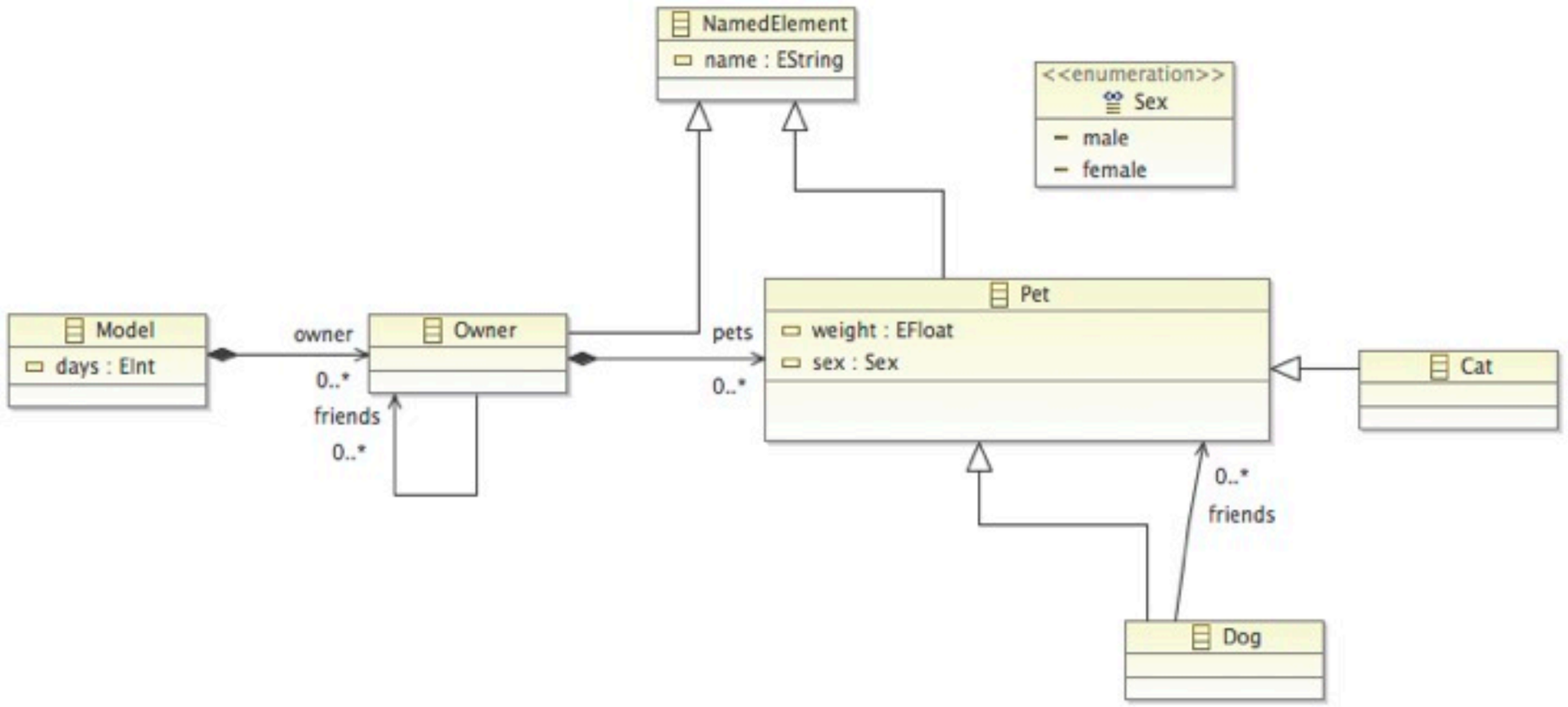



```
private State createState(String name, MarkovModel model) {
    State state = MarkovFactory.eINSTANCE.createState();
    state.setName(name);
    model.getStates().add(state);
    return state;
}

private Transition createTransition(State from, State to, float propability) {
    Transition transition = MarkovFactory.eINSTANCE.createTransition();
    from.getOutgoingTransitions().add(transition);
    transition.setPropability(propability);
    transition.setTarget(to);
    return transition;
}
```

Another Example

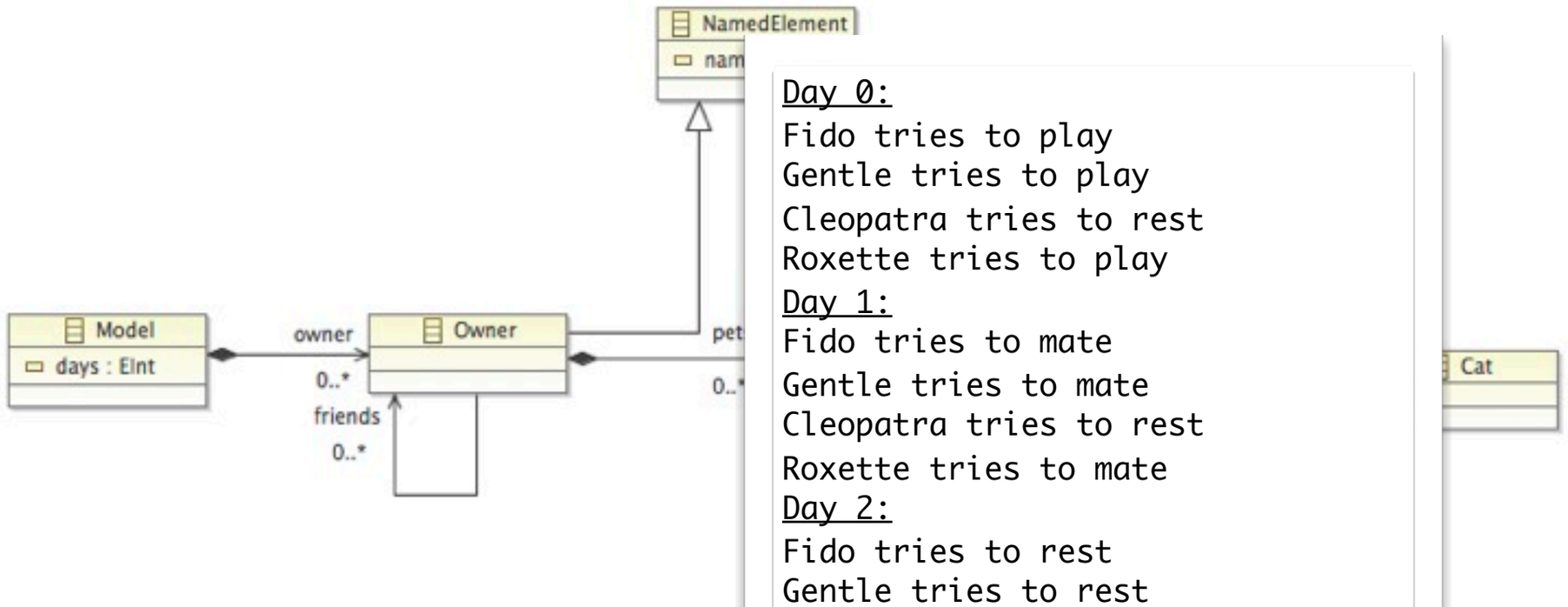


Another Example




```
Markus is friends with Kathi and owns
    male dog Fido, male cat Gentle

Kathi owns
    female dog Cleopatra, female cat Roxette
```

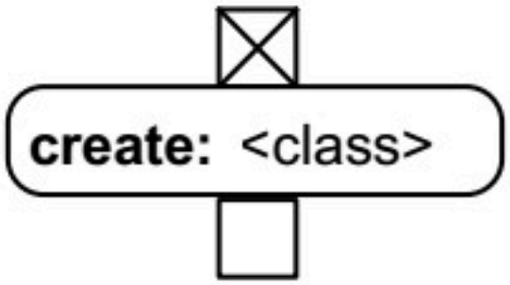
Another Example



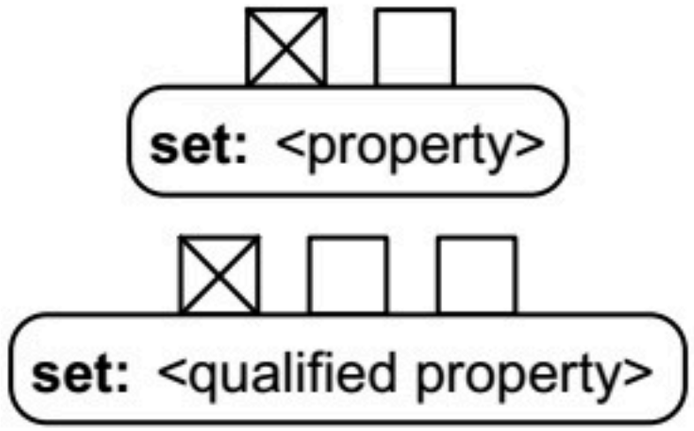

```
Markus is friends with Kathi and owns
    male dog Fido, male cat Gentle

Kathi owns
    female dog Cleopatra, female cat Roxette
```

```
Day 0:
Fido tries to play
Gentle tries to play
Cleopatra tries to rest
Roxette tries to play
Day 1:
Fido tries to mate
Gentle tries to mate
Cleopatra tries to rest
Roxette tries to mate
Day 2:
Fido tries to rest
Gentle tries to rest
Cleopatra tries to rest
Roxette tries to rest
Day 3:
Fido tries to play
Gentle tries to play
Cleopatra tries to rest
Roxette tries to play
Day 4:
Fido tries to mate
Gentle tries to mate
Cleopatra tries to rest
Roxette tries to mate
Day 5:
Fido tries to rest
Gentle tries to rest
Cleopatra tries to rest
Roxette tries to rest
```

Action Language

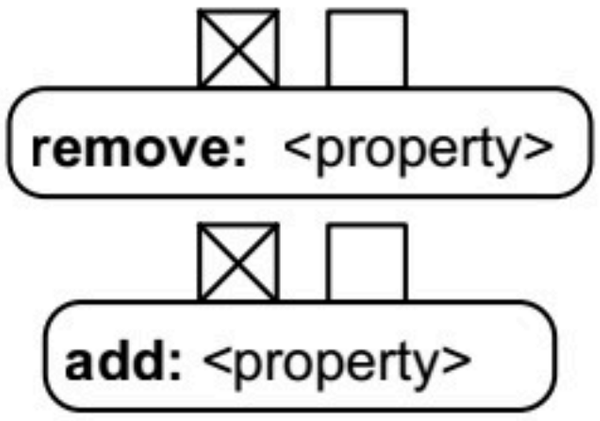


Creates a
new instance.



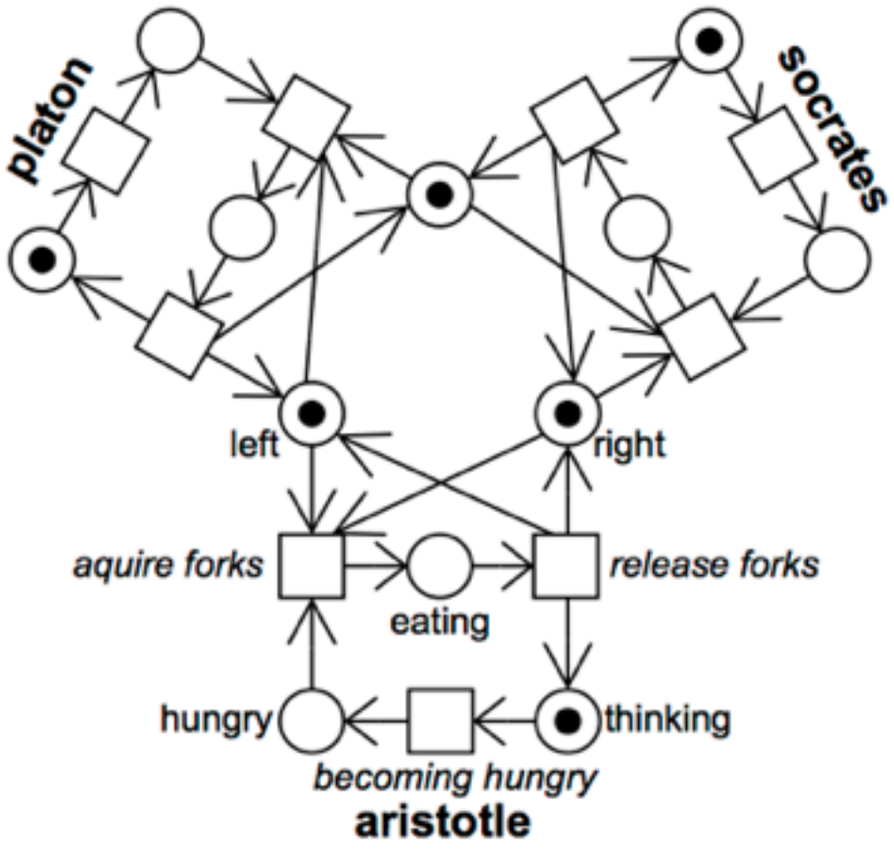
Sets the value of a property with multiplicity 1 or 0..1. For properties with qualifiers additional input pins are required

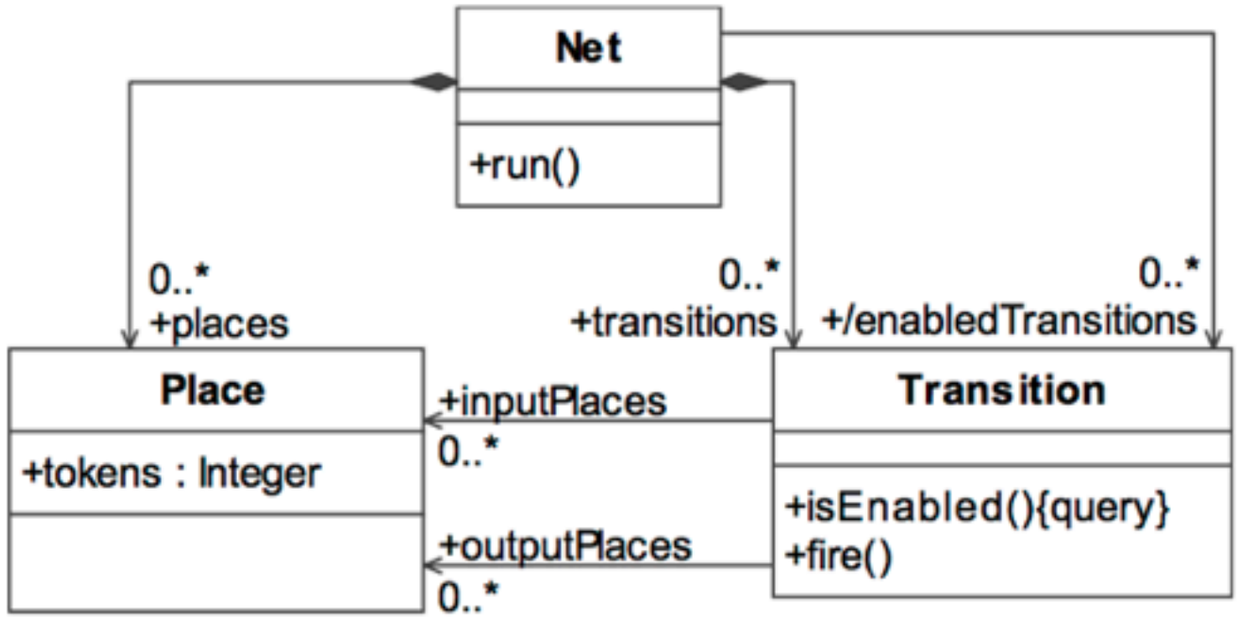
---



Adds/removes an value to/from a property with multiplicity bigger than one.

Action Language Example – Petri Nets

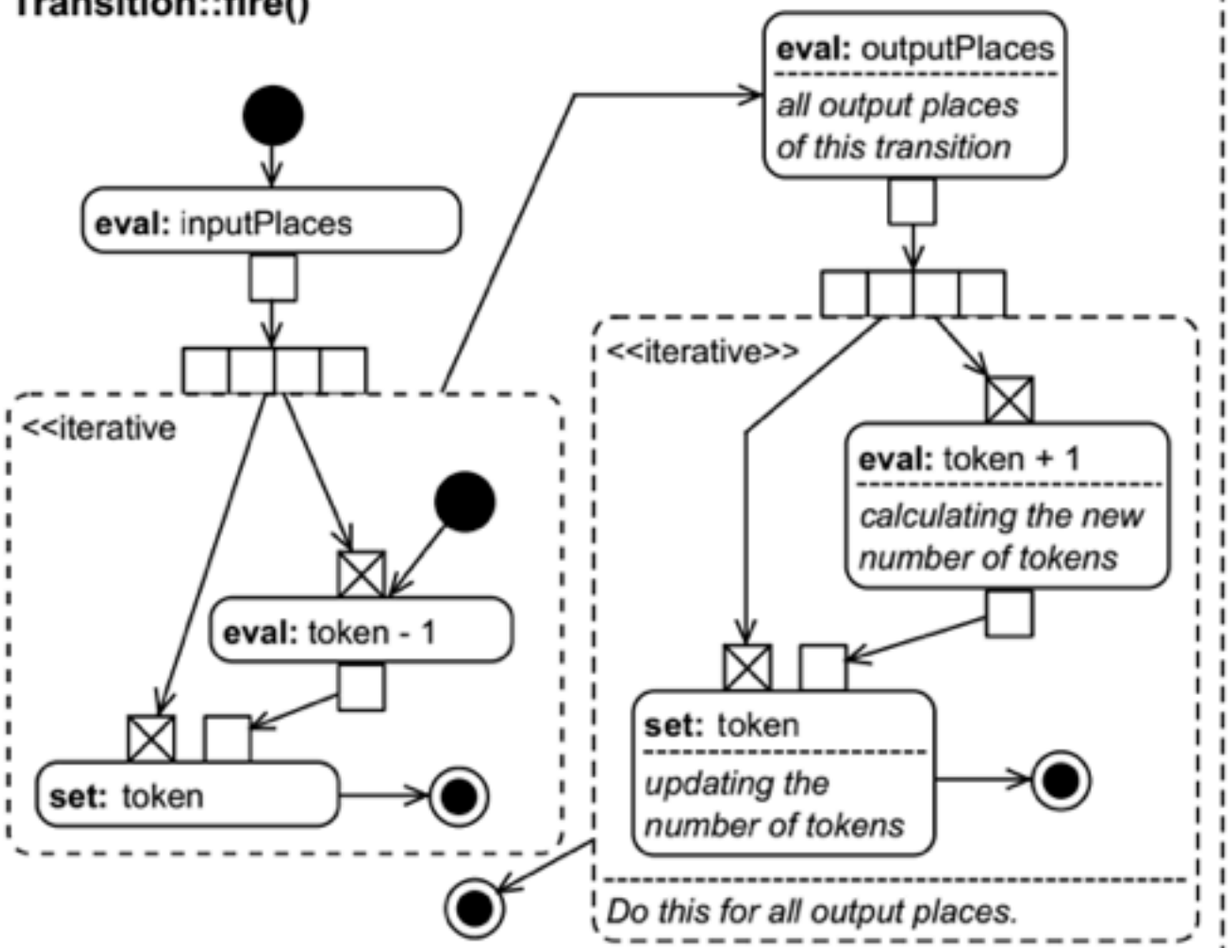




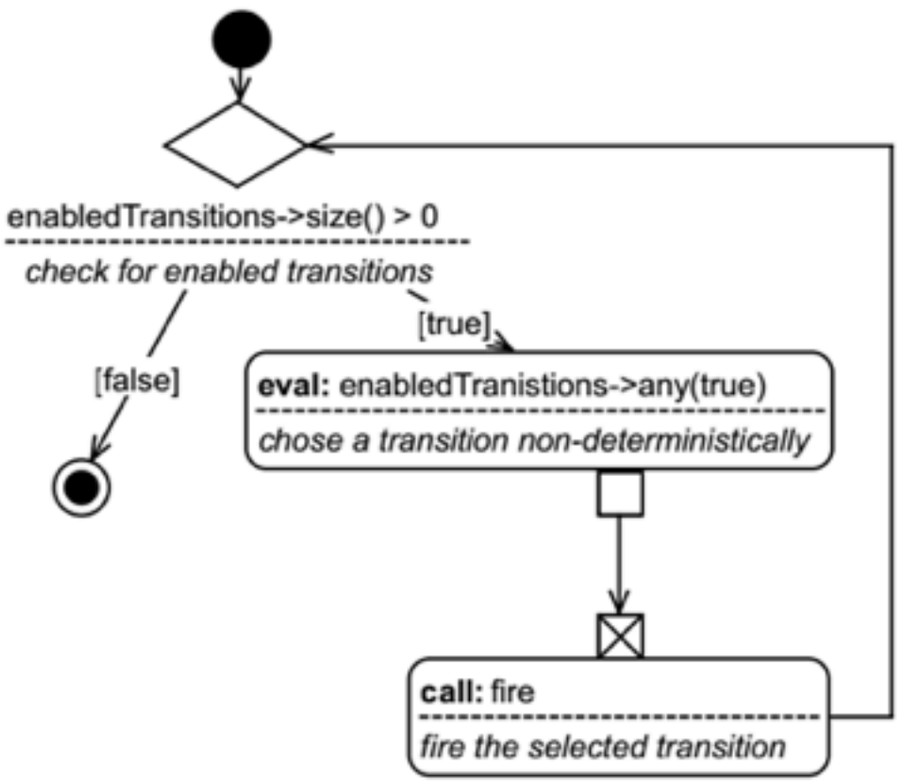
context Net::enabledTransitions
derive:
transitions->select(isEnabled())

context Transition::isEnabled()
body:
inputPlaces->forAll(tokens>0)

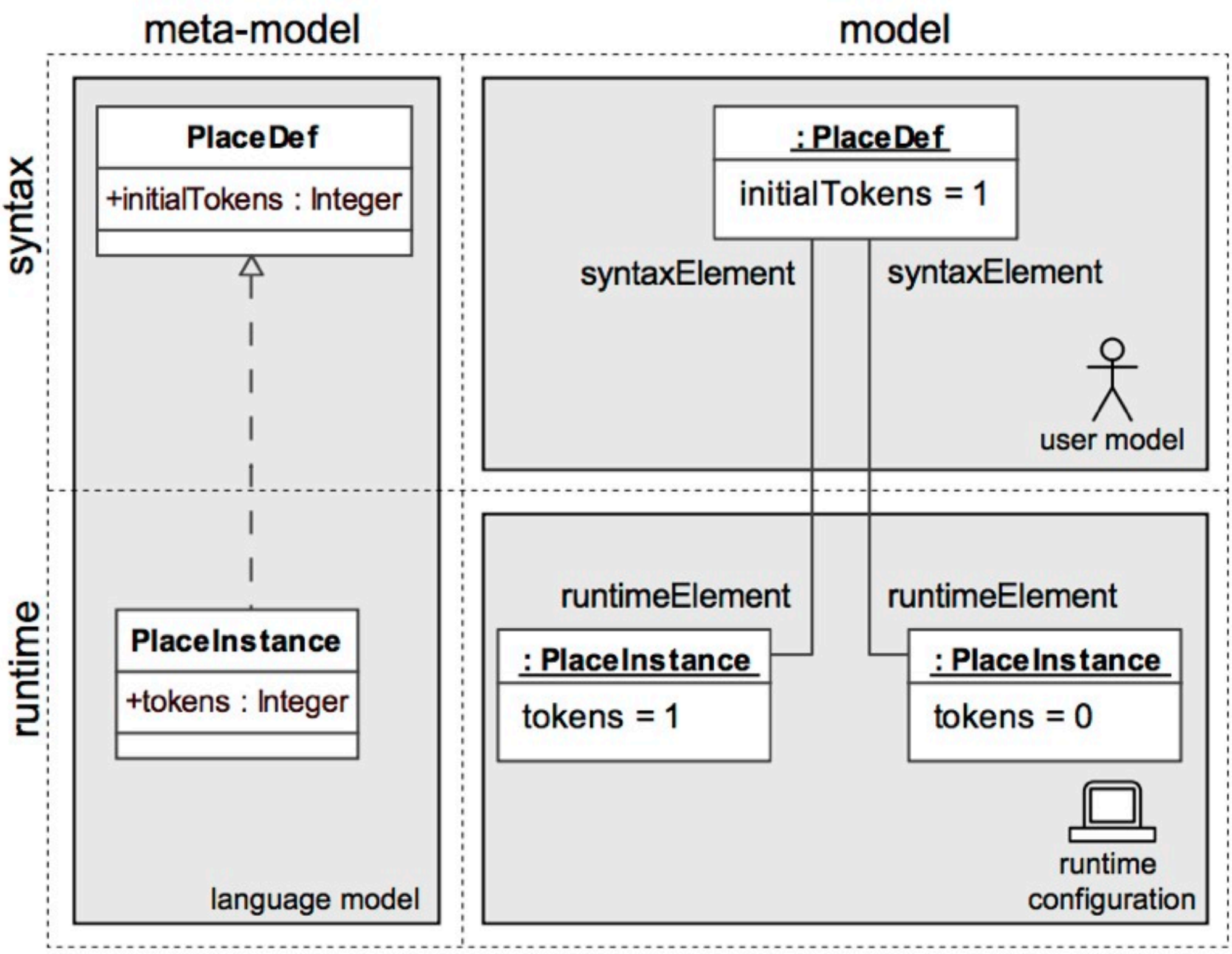
Transition::fire()



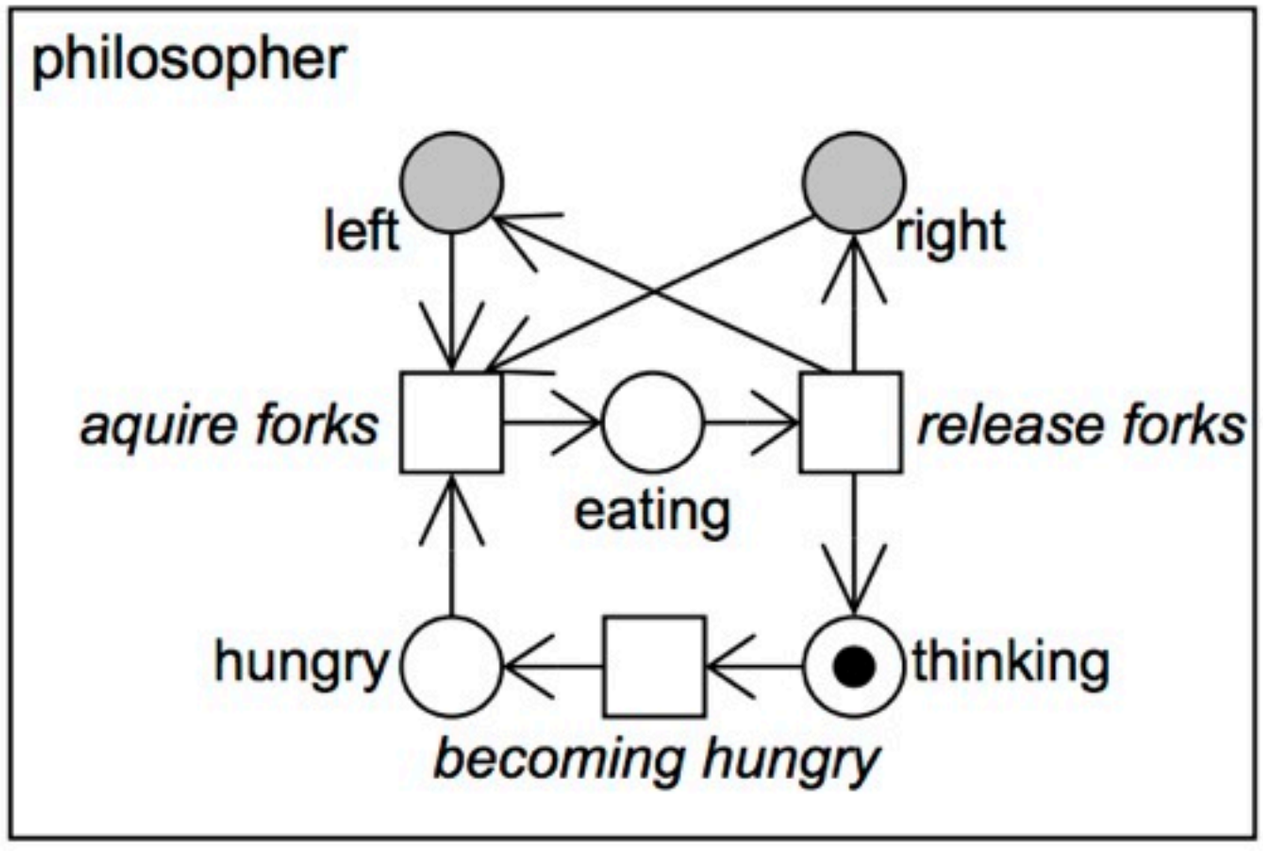
Net::run()

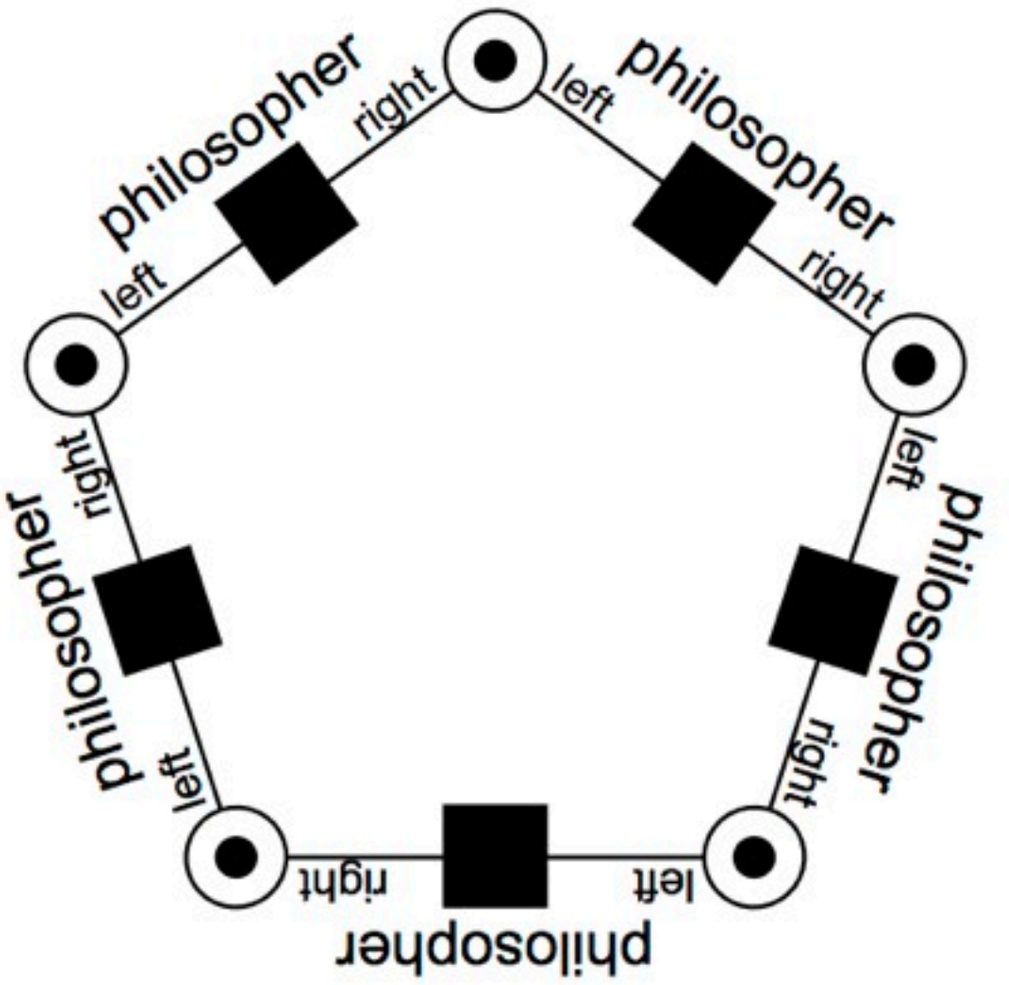


Abstract Syntax and Runtime Concepts

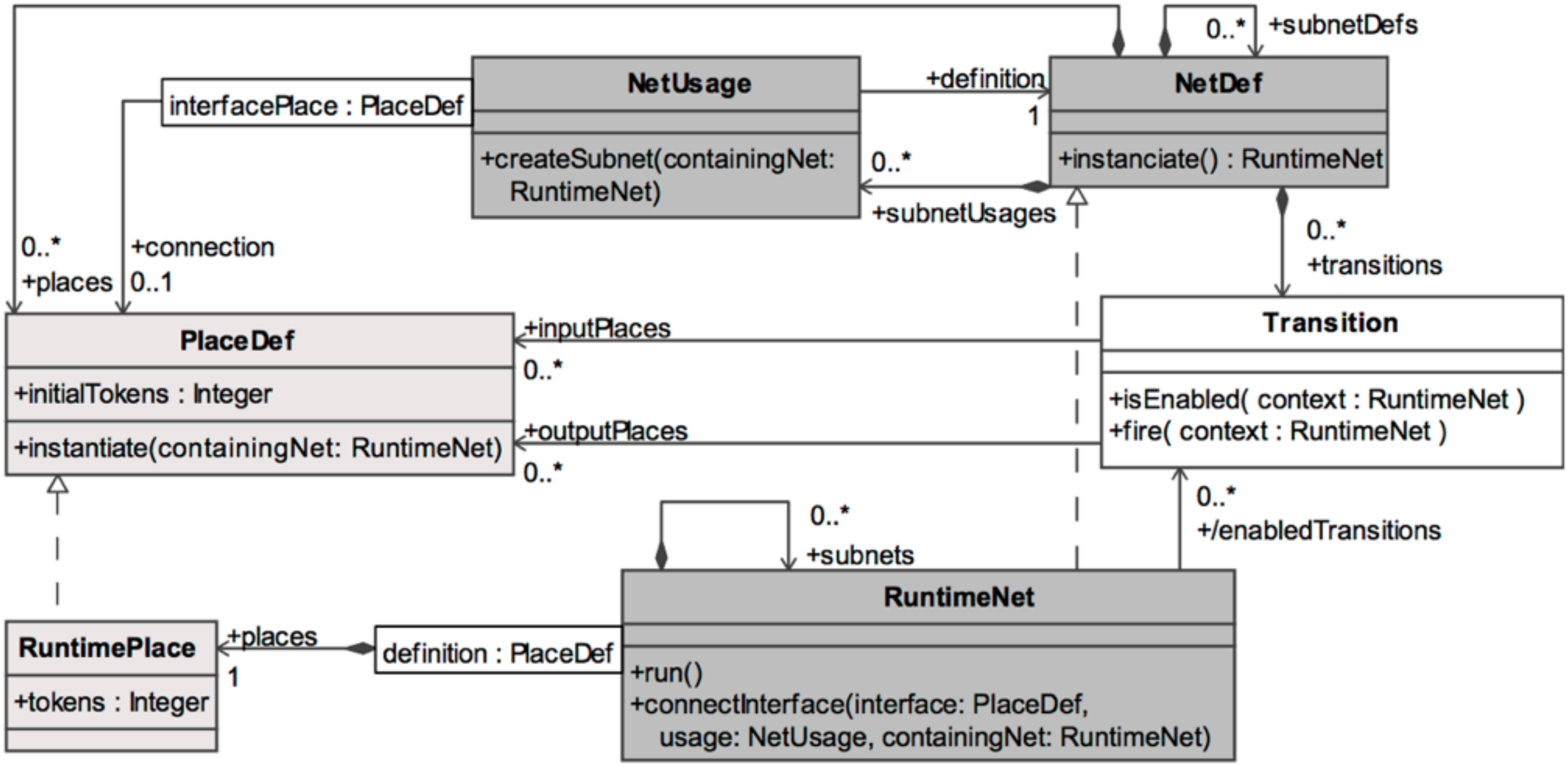


Runtime Concepts – Example Hierarchical Petri Nets





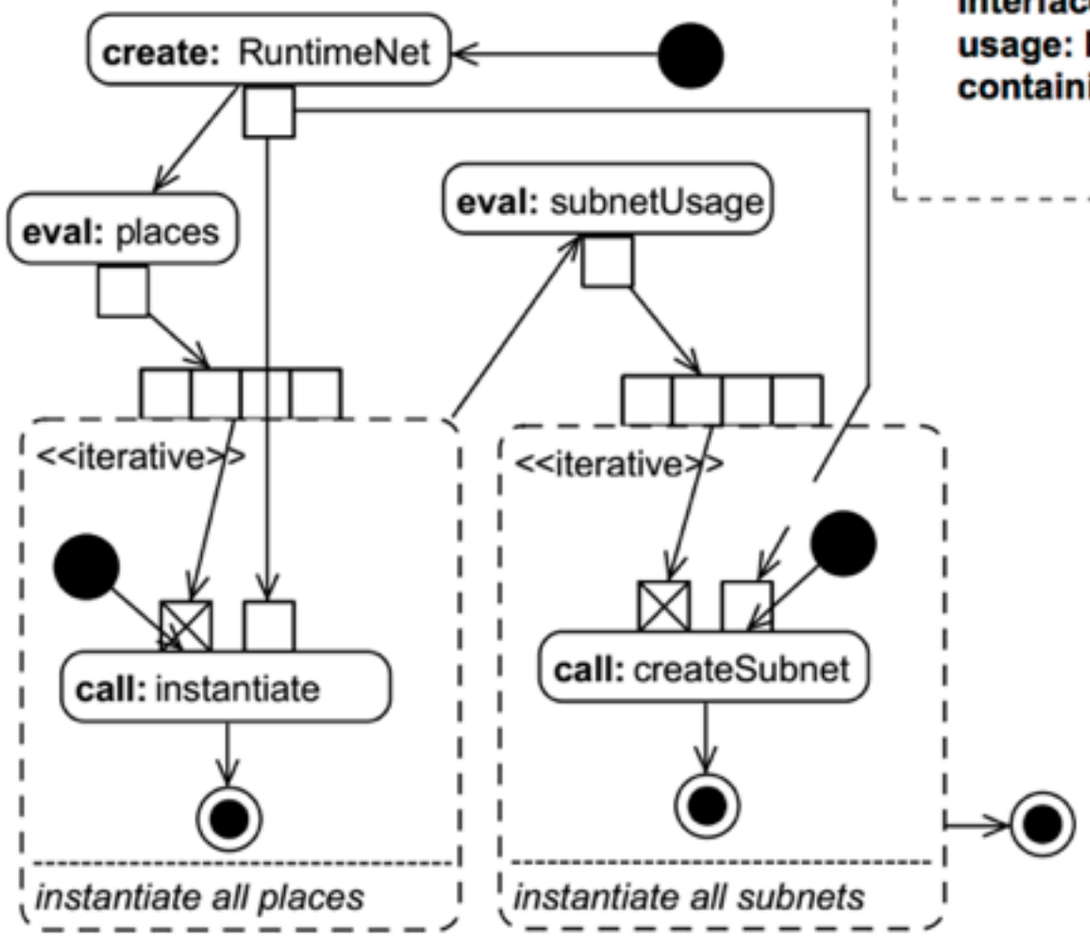
Runtime Concepts – Example Hierarchical Petri Nets



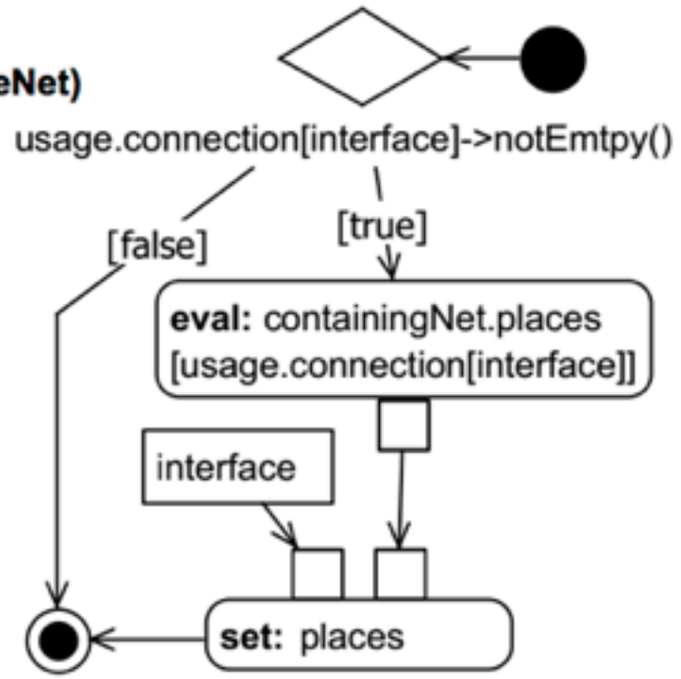
context RuntimeNet::enabledTransitions
derive:
subnets->collect(enabledTransitions)->union(
syntaxElement.transitions->select(isEnabled(self))

context Transition::isEnabled(context: RuntimeNet)
body:
inputPlaces->forAll(ip|context.places[ip].tokens > 0)

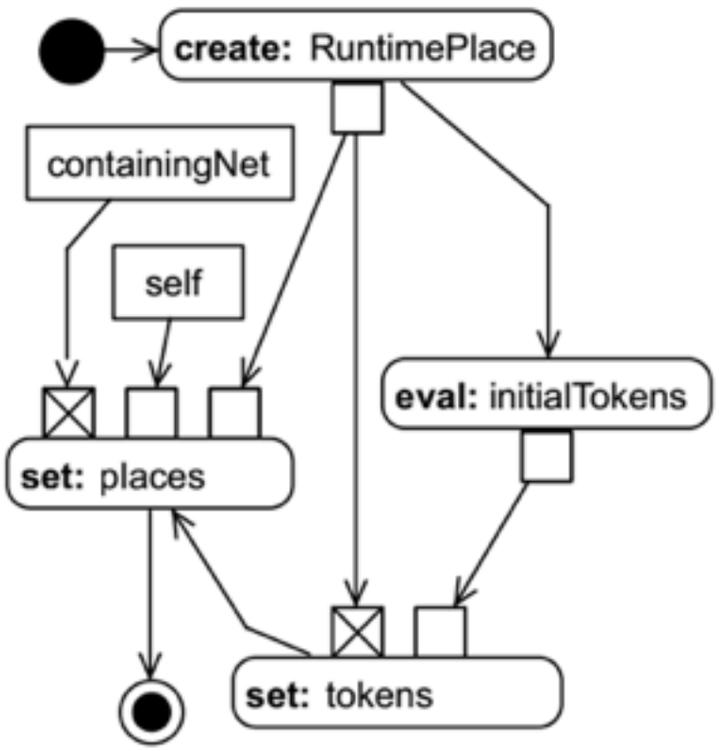
NetDef::instantiate()



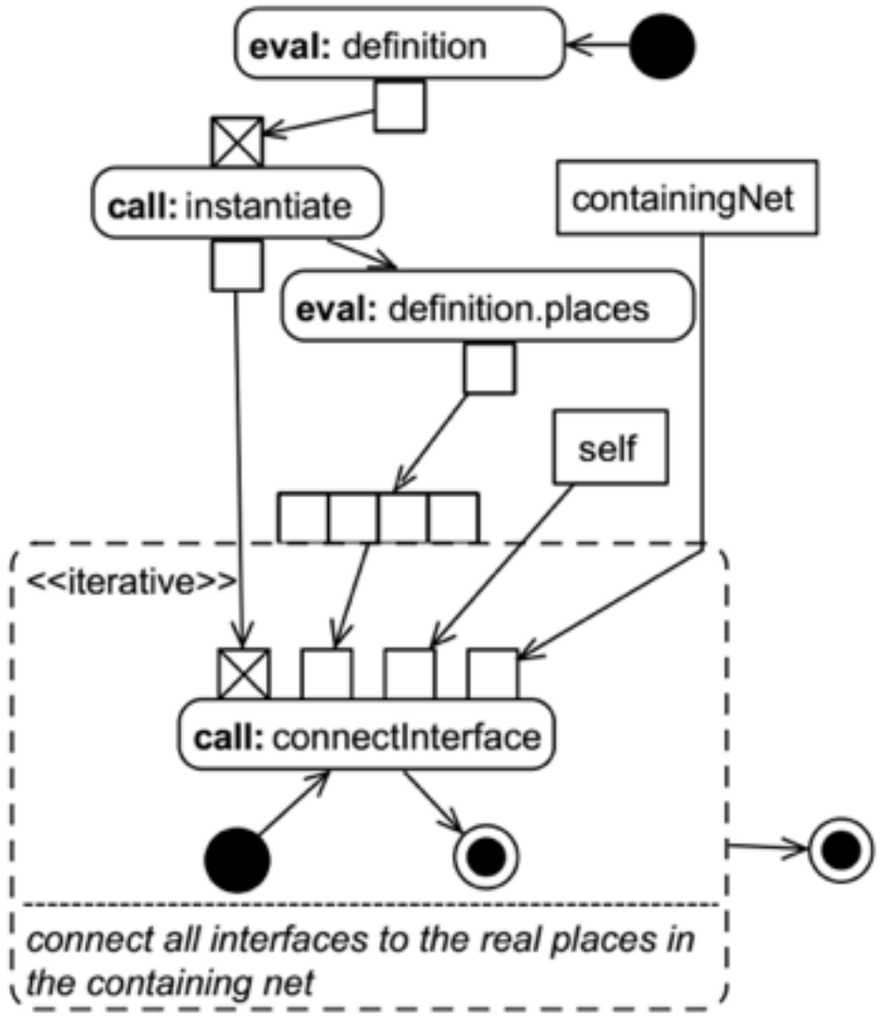
RuntimeNet::connectInterface(
interface: PlaceDef,
usage: NetUsage,
containingNet: RuntimeNet)



PlaceDef::instantiate(containingNet: RuntimeNet)



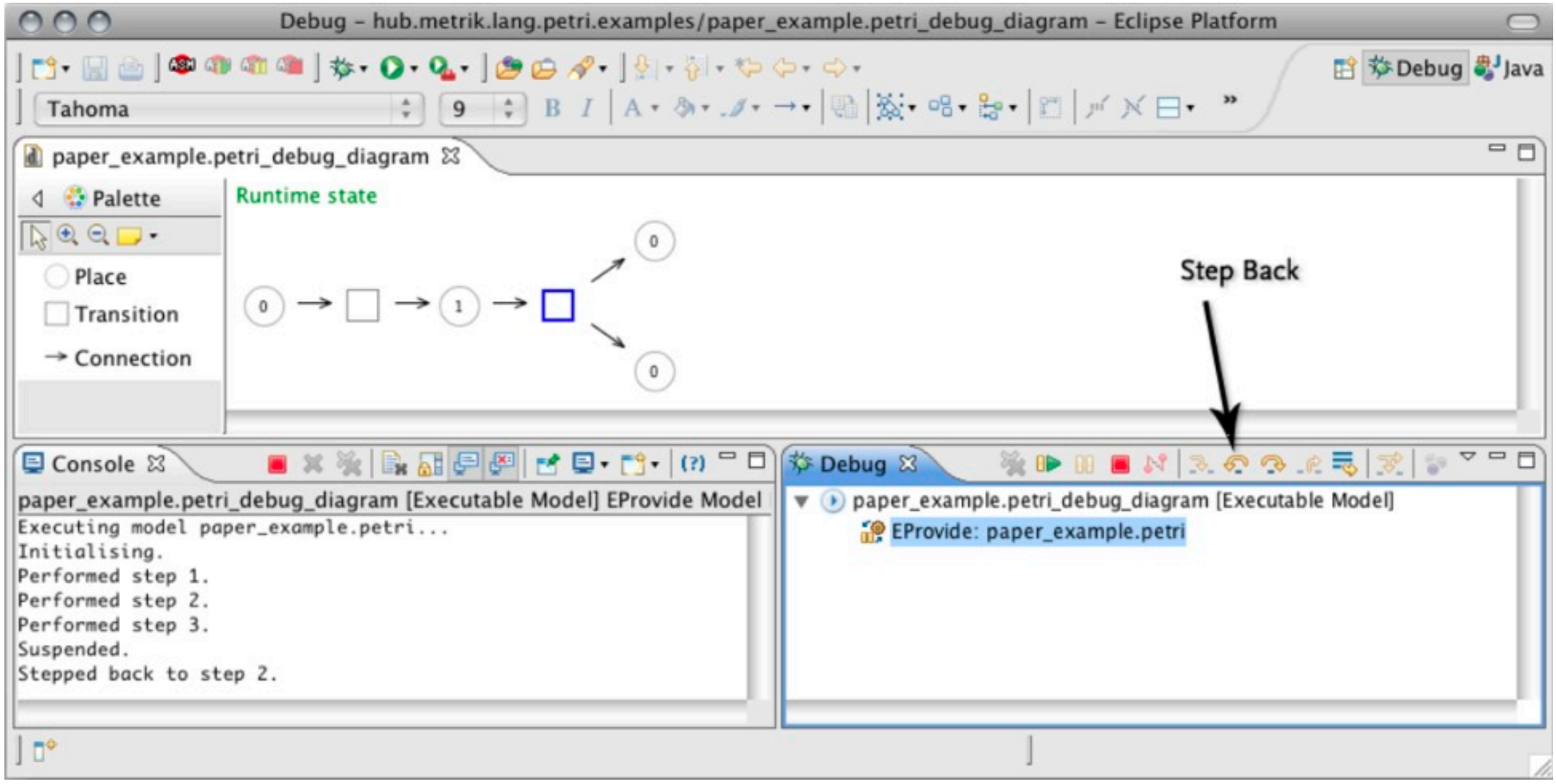
NetUsage::createSubnet(containingNet: RuntimeNet)



Traces and Debugging

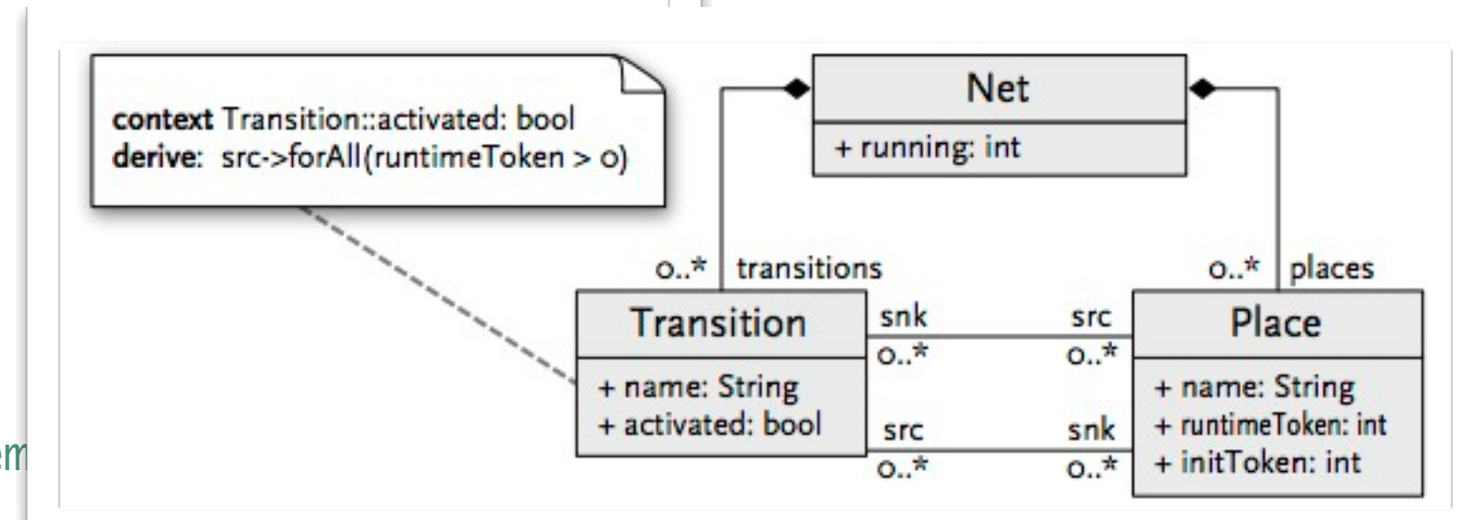
- Only actions change the model
- It's good practice to only modify the runtime-part of a model and retain the user model/program
- Actions can be recorded as traces of the execution
  - reverse actions to go backwards
- Intermediate models can be stored (compare heap dump in traditional programming)
- generated EMF edit and notifications can be used to create views on the runtime for a custom debugger
- no easy out of the box debugging
  - no separation between model/program and semantics description

Traces and Debugging



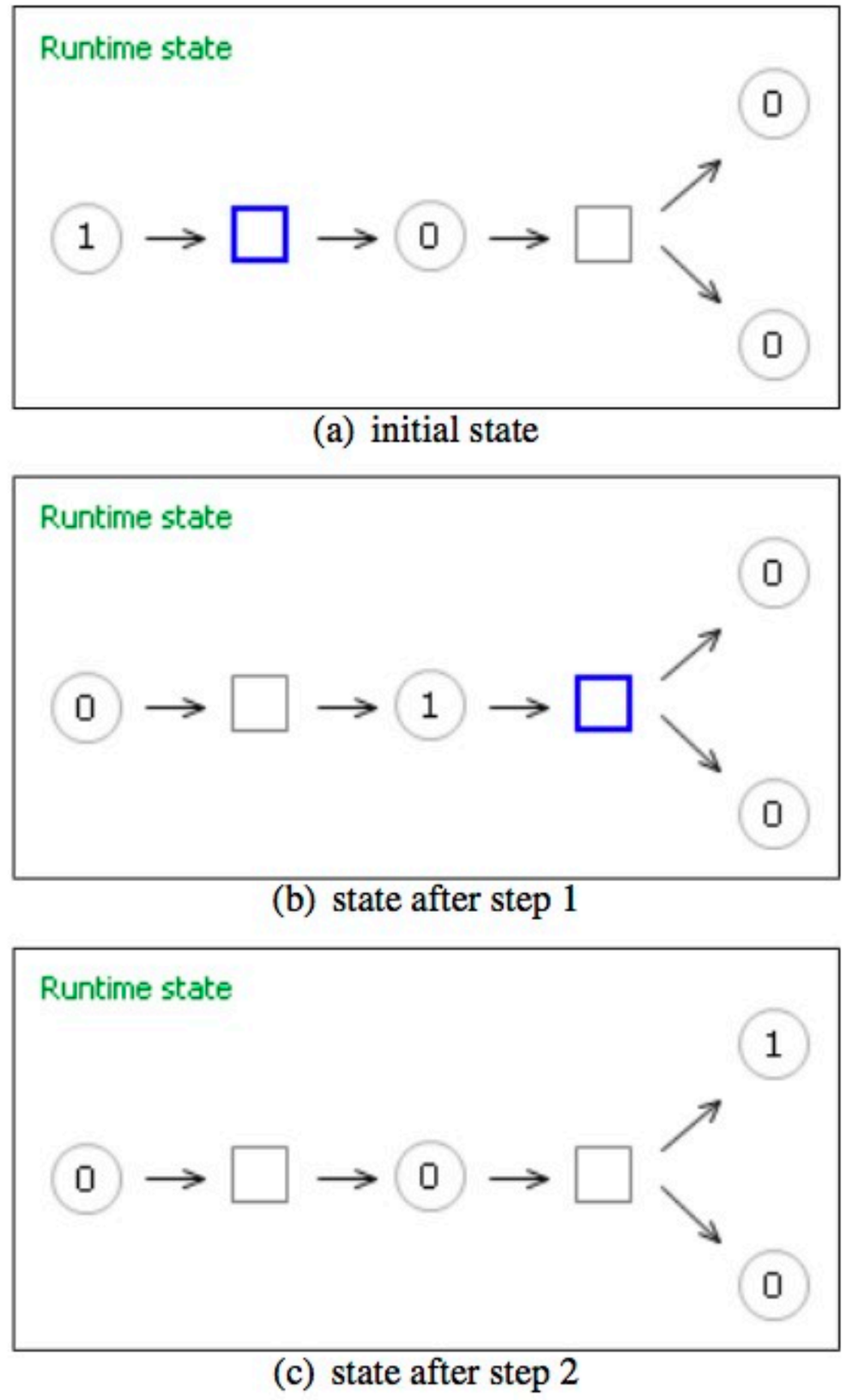
Traces and Debugging

```
public class PetriSemantics implements ISemanticsProvider {
    public void step(Resource model) {
        Net net = (Net) model.getContents().get(0);
        net.setRunning(true);
        fireTransition(net);
    }

    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransitions(net);
        if (!ats.isEmpty()) {
            Transition t = choose(ats);
            Place p = choose(t.getSrc());
            consume(p);
            p = choose(t.getSnk());
            produce(p);
        }
    }

    protected T choose(List<T> list) {
        // Returns a randomly chosen mem
    }
    // ...
}
```



Traces and De

```
public class PetriSemantics implements ISemanticsP
    public void step(Resource model) {
        Net net = (Net) model.getContents().get(0);
        net.setRunning(true);
        fireTransition(net);
    }

    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransi
        if (!ats.isEmpty()) {
            Transition t = choose(ats);
            Place p = choose(t.getSrc());
            consume(p);
            p = choose(t.getSnk());
            produce(p);
        }
    }

    protected T choose(List<T> list) {
        // Returns a randomly chosen mem
    }
    // ...
}
```



(a) initial state

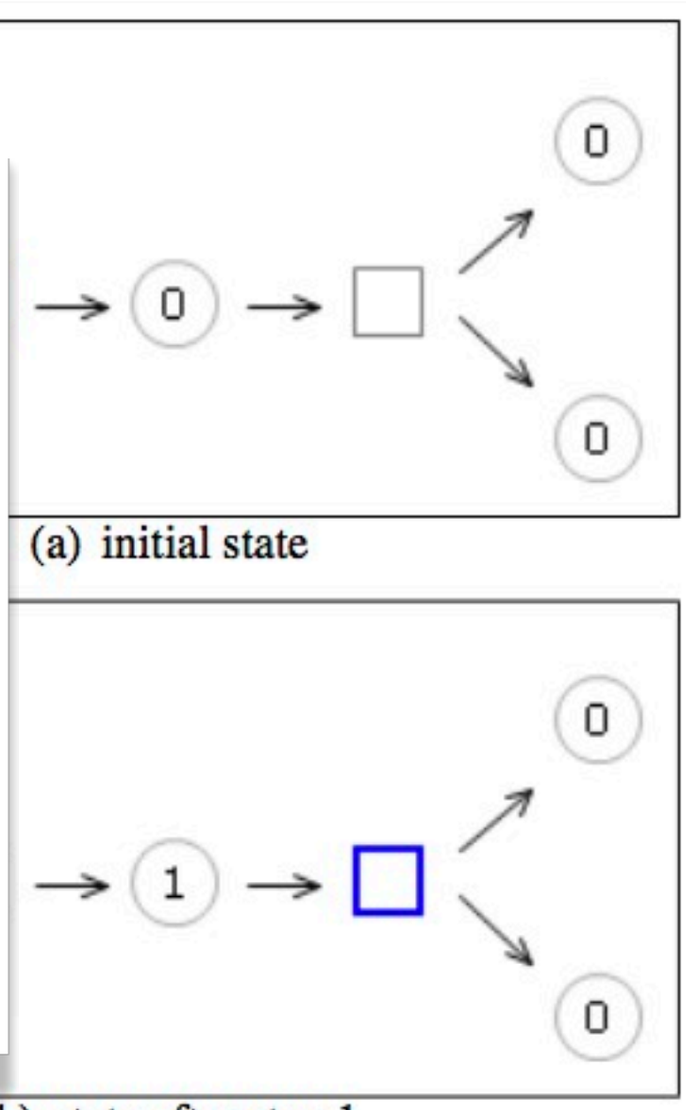
(b) state after step 1

(c) state after step 2

Traces and De

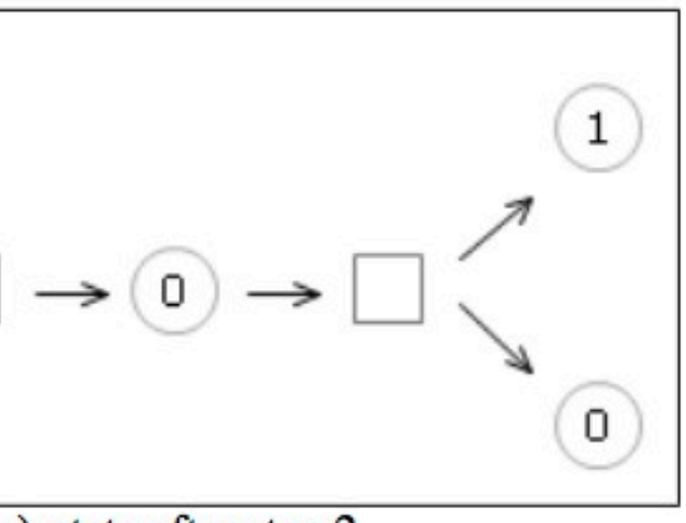
```
    // ...
    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransitions(net);
        if (!ats.isEmpty()) {
            Transition t = choose(ats);
            for (Place p : t.getSrc())
                consume(p);
            for (Place p : t.getSnk())
                produce(p);
        }
    }
```

```
        // ...
            consume(p);
            p = choose(t.getSnk());
            produce(p);
        }
    }

    protected T choose(List<T> list) {
        // Returns a randomly chosen mem
    }
    // ...
}
```



(a) initial state

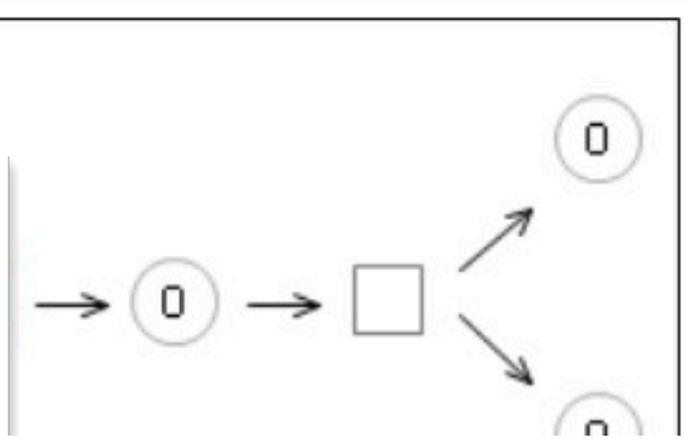
(b) state after step 1

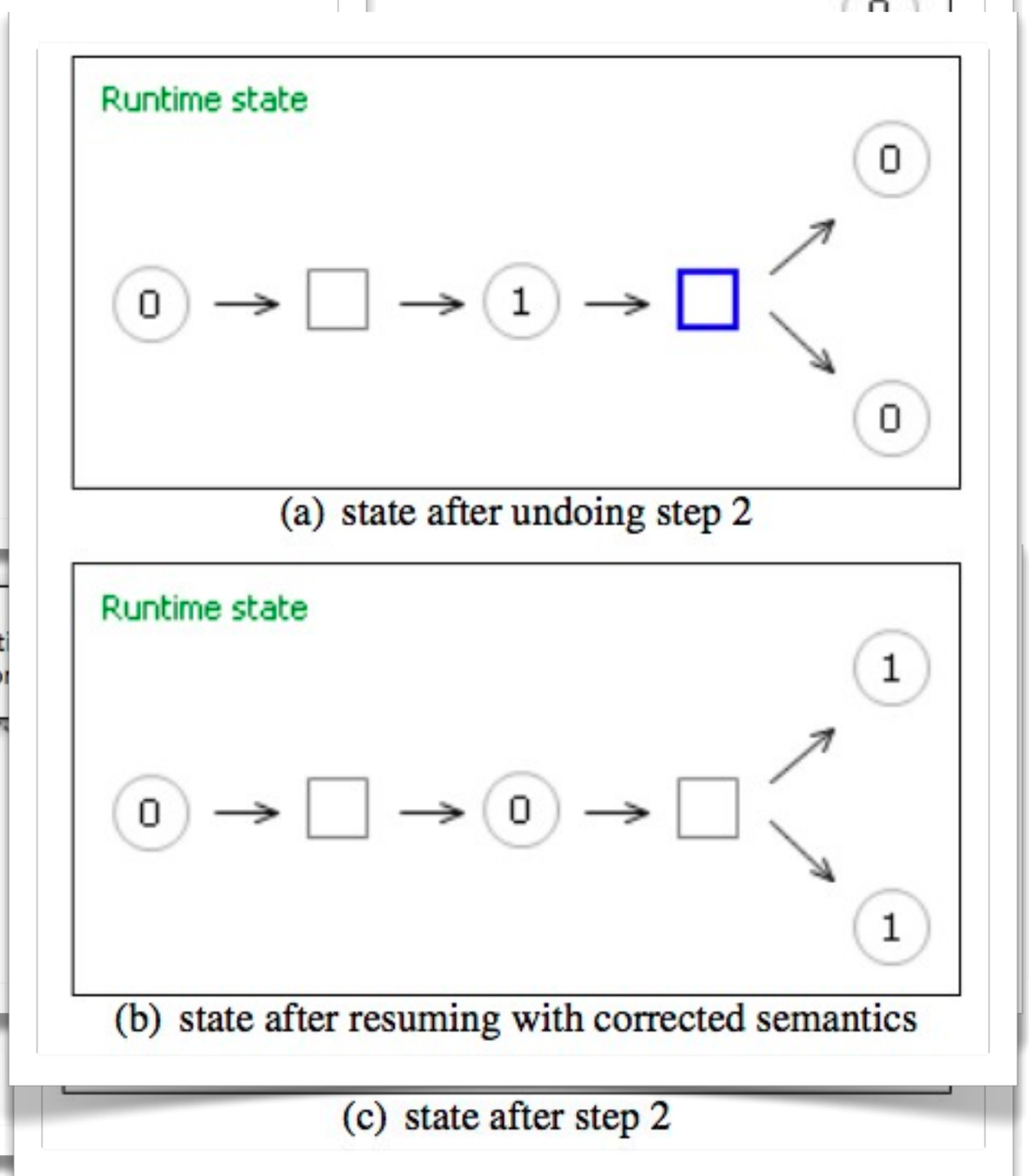


(c) state after step 2

Traces and De

```
    // ...
    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransitions(net);
        if (!ats.isEmpty()) {
            Transition t = choose(ats);
            for (Place p : t.getSrc())
                consume(p);
            for (Place p : t.getSnk())
                produce(p);
        }
    }

    // ...
            consume(p);
            p = choose(t.getSnk());
            produce(p);
        }
    }

    protected T choose(List<T> list) {
        // Returns a randomly chosen mem
    }
    // ...
}
```





Using the Environment

- Reasonable models/programs need to interact with the environment when simulated/run
  - Input/output
  - Interaction with eclipse or other GUI elements
  - Interaction with databases
  - Simulation visualization
  - ...
- EMF is not self-contained: use operations and datatypes to connect EMF to the rest of the Java world
- notations can be used to visualize runtime state

Summary

- Add runtime-concepts to the meta-model
- Declare operations
- Implement operations, e.g. with Java or M3Actions
- Interpreters need to load the model/program and call the *main* operation.
- Lots of possibilities to debug and to build custom debuggers, no simple out of the box solution

Translational Semantics

with EMF

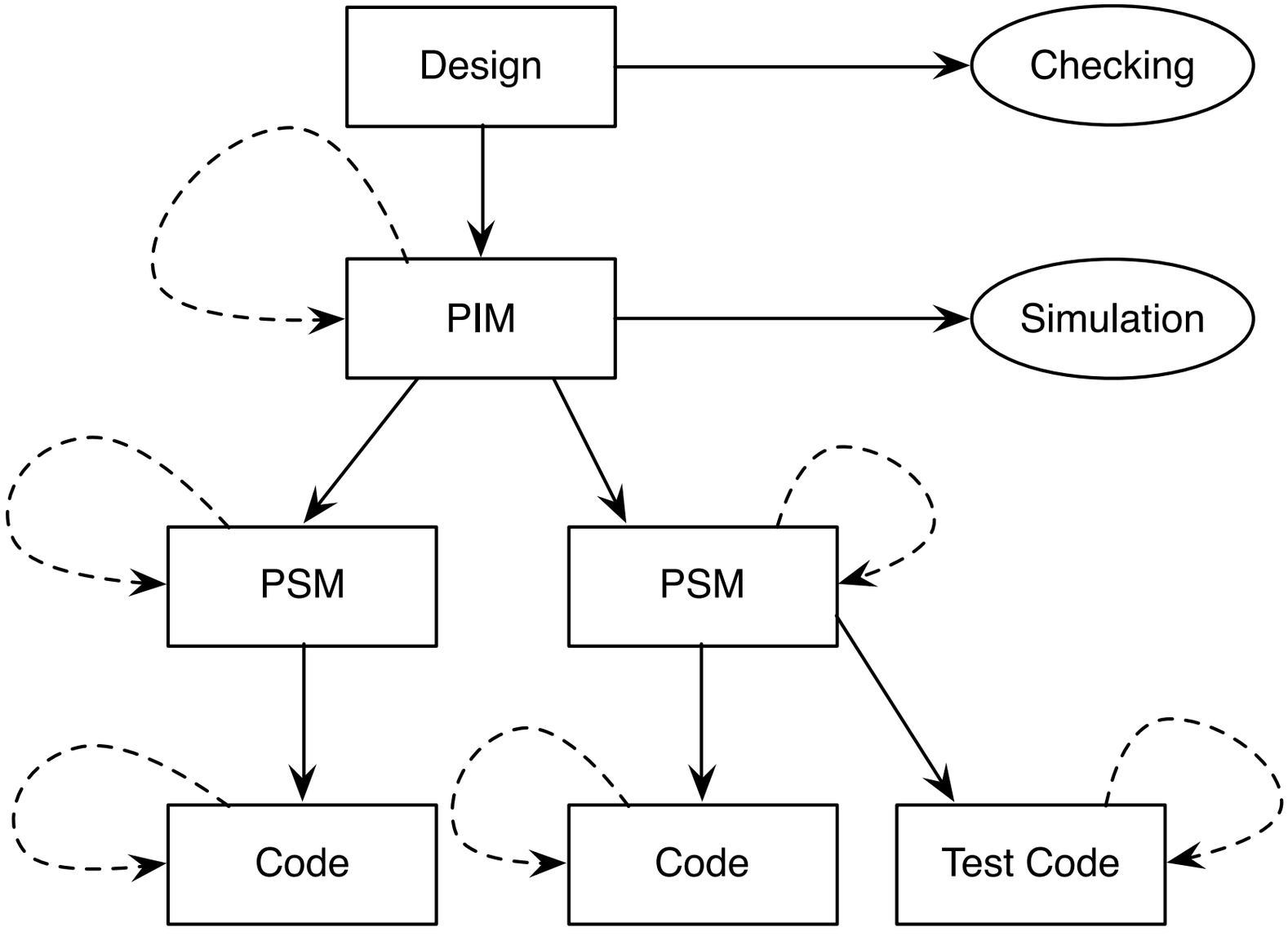
Types of “Model Transformations”

- Operational semantics
  - Interpretation (model-to-execution)
- Translational semantics
  - Code-generation (model-to-code)
  - Model-transformation (model-to-model)
    - new target
    - existing target
    - source=target, in-place transformation
    - further classification necessary

Elaboration and Translational Semantics

- Generated artifacts can be modified or extended after generation
- Elaboration allows to vary the generated semantics, i.e. allows variance in the semantics description
- Generated code can be modified, generated models can be extended
- Elaboration is paramount for practical abstraction
  - more flexibility for language users
  - smaller, more coherent, less expensive DSLs for language engineers
  - mitigates some problems of external DSLs (when compared to internal DSLs)
- Elaboration and re-generation
  - protected regions
  - elaboration by extension, if the target language supports external extension of completed entities like e.g. in most object-oriented languages

Elaboration and Translational Semantics



Translational Semantics with EMF

- Programming
  - Java or other JRE compatible languages
  - other programming languages via XMI/XML
- Languages for code-generation
  - templates, e.g. Jet
  - programming languages with rich-strings, e.g. xtend
- Languages for model-to-model transformations
  - imperative, e.g. ATL
  - declarative, e.g. (triple graph) grammars

Operational vs. Translational

- self-contained
- requires a specific runtime environment almost all the time
- debuggable
- platform specific, requires model processing on that platform
- interpreters can be parameterized for semantic variations
- no generated artifacts, no elaboration of generated artifacts
- no generated artifacts that need to be maintained

- target language dependent
- sometimes requires specific runtime environment
- hard to debug
- “platform independent”, platform does not need to process model
- model transformations can be parameterized for semantic variations
- generated code can be elaborated for semantic variations
- generated code is another asset to maintain

Code-Generation vs. Model-Transformartions

- No guaranties that generated artifacts are well-formed or even semantically sound
- In general, no properties can be formally proved
- Structural differences between source and target possible
- Generated artifacts can be syntactically elaborated (there is concrete syntax)
- generated artifacts are at least syntactically sound (no concrete syntax involved)
- In theory and for some techniques, some properties (e.g. retention of properties) can be proved
- Its harder to create structurally different targets with most model transformation languages
- Elaboration of generated artifacts only via external extension

Translational Semantics

Code Generation

Approaches to Code Generation with EMF

- Programming
  - Java or other JRE compatible languages (e.g. Groovy, Scala)
  - other languages via XMI/XML
- Template Language
  - XML-based languages, e.g. XSLT
  - EMF-based languages, e.g. Jet
- Rich Strings
  - Programming languages that support *Rich Strings*, e.g. xTend

Programming Code Generators

- programmatically traverse and navigate the model, e.g. via higher order collection functions, internal OCL-like DSLs
- basic IO to *print* code snippets while traversing the model
- generated code can use runtime concepts implemented in a runtime library.
- the generated code can be written in another language or even in no formal/computer language at all

Java Example For Code Generator

```
public void generate(Model model) {
    for(Owner owner: model.getOwner()) {
        for (Pet pet: owner.getPets()) {
            System.out.println(pet.getName() + "Rest");
            System.out.println("  to " + pet.getName() + "Play with " + (pet.getPlayPartners(
            System.out.println("  to " + pet.getName() + "Mate with " + (pet.getMatePartners(
            System.out.println("");


            System.out.println(pet.getName() + "Mate");
            System.out.println("  to " + pet.getName() + "Rest with 1");
            System.out.println("");

            System.out.println(pet.getName() + "Play");
            System.out.println("  to " + pet.getName() + "Mate with " + Math.pow(Math.E, -(1/
            System.out.println("  to " + pet.getName() + "Mate with " + (1-Math.pow(Math.E, -
            System.out.println("");
        }
    }
}
```

Java Example For Code Generator

```
+ (pet.getPlayPartners()/(float)(pet.getPlayPartners()+pet.getMatePartners())));
+ (pet.getMatePartners()/(float)(pet.getPlayPartners()+pet.getMatePartners())));

');


+ Math.pow(Math.E, -(1/(float)pet.getMatePartners())));
+ (1-Math.pow(Math.E, -(1/(float)pet.getMatePartners()))));
```

Java Example For Code Generator

```
Markus is friends with Kathi and owns
     male dog Fido, male cat Gentle

Kathi owns
     female dog Cleopatra, female cat Roxette

Peter owns
     female dog Susi, male cat Victor
```

```
FidoRest
  to FidoPlay with 1.0
  to FidoMate with 0.0

FidoMate
  to FidoRest with 1

FidoPlay
  to FidoMate with 0.0
  to FidoMate with 1.0

GentleRest
  to GentlePlay with 0.5
  to GentleMate with 0.5

GentleMate
  to GentleRest with 1

GentlePlay
  to GentleMate with 0.36787944117144233
  to GentleMate with 0.6321205588285577

CleopatraRest
  to CleopatraPlay with 0.0
  to CleopatraMate with 1.0
```

Rich Strings

- Part of some programming languages
- “printf”-style code generation, but with better “richer” strings that support
  - indentation
  - control sequences
  - model expressions
  - method calls

Example Rich String Code Generator (xTend)

```
def generate(Model model) '''
    «FOR pet:model.owner.fold(new ArrayList<EList<Pet>>)[r,o|r.addAll(o.pets); r].flatten»
        «pet.name»Rest
            to «pet.name»Play with «pet.playPartners/((pet.playPartners+pet.matePartners) as float)»
            to «pet.name»Mate with «pet.matePartners/((pet.playPartners+pet.matePartners) as float)»

        «pet.name»Mate
            to «pet.name»Rest with 1

        «pet.name»Play
            to «pet.name»Mate with «Math.pow(Math.E, -(1/(pet.matePartners as float)))»
            to «pet.name»Rest with «1-Math.pow(Math.E, -(1/(pet.matePartners as float)))»
    «ENDFOR»
'''
```

Templates

- Rich strings allow to embed generated code into the code-generator code
- Templates allow to embed code-generator code into the generated code
- similar approach to JSP or PHP
- e.g. with JET (subset of JSP), *Java Emitter Templates*
  - model expressions (Java)
  - control sequences (Java)
  - calls to other templates (JET)
  - embedded code is Java code

Template Example

```
<%@ jet
    package="fido"
    imports="de.hub.sam.modsoft.fido.*"%>
<% Model model = (Model) argument; %>
<%
for(Owner owner: model.getOwner()) {
    for (Pet pet: owner.getPets()) {
%>
<%=pet.getName()%>Rest
     to <%=pet.getName()%>Play with <%=pet.getPlayPartners()/(float)(pet.getPlayPartners()+p
     to <%=pet.getName()%>Mate with <%=pet.getMatePartners()/(float)(pet.getPlayPartners()+p

<%=pet.getName()%>Mate
    to <%=pet.getName()%>Rest with 1


<%=pet.getName()%>Play
    to <%=pet.getName()%> with <%=Math.pow(Math.E, -(1/(float)pet.getMatePartners()))%>
    to <%=pet.getName()%> with <%=1-Math.pow(Math.E, -(1/(float)pet.getMatePartners()))%>
<%
    }
}
%>
```

xTend and JET Semantics

- How is xTend and JET Semantics realized?
  - parser plus code generator to Java
  - generated Java code uses “printf”-style code generation
  - xTend is a full programming language with syntax and static semantics
  - JET just embeds Java snippets that are not directly checked for correct syntax and static semantics

Code Generation and Elaboration

- Protected regions
  - specifically marked regions in the generated code are not regenerated
  - positive and negative
  - required target syntax to contain marks (comments, annotations)
  - e.g. *generated NOT* in EMF
  - required maintenance of generated code (e.g. version control)
- Generation gap pattern
  - generated classes are extended and functionality is altered via callbacks and overwriting
  - only works for object oriented target language
  - can only change what is meant to be changed
  - full separation of generated and not generated code (generation gap), generated code does not need to be maintained (e.g. version controlled)

Summary

- Rich strings (e.g. in Xtend) and Template languages can be used for code-generation
- Code-generation description language/framework is independent of the target language
- very flexible, but unsafe
- protected regions vs. generation gap pattern to elaborate generated code