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*

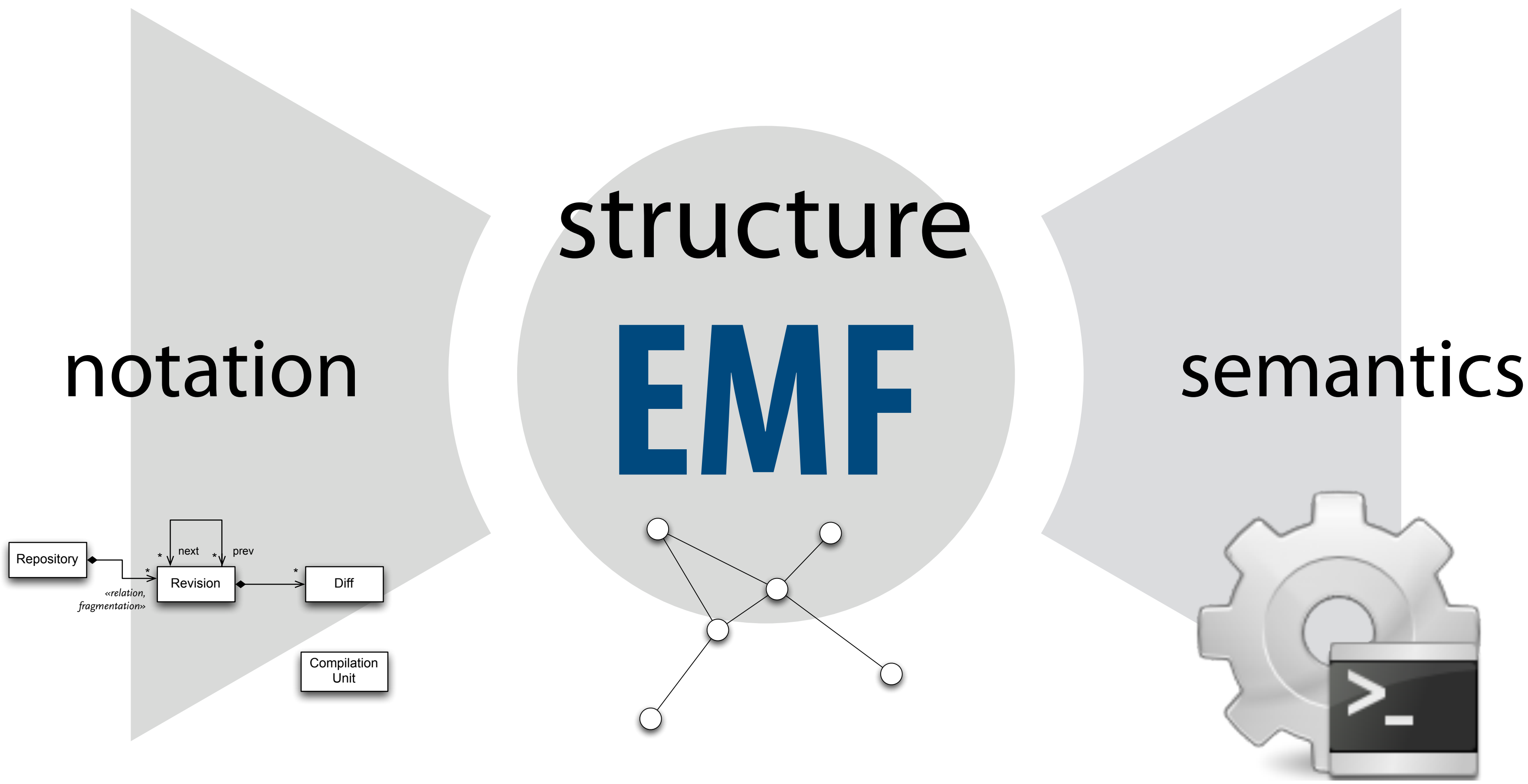
2. (3 VL) — **Notation:** Customizing the tree-editor, textural with *XText*, graphical with *GEF* and *GMF*

➡ 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)

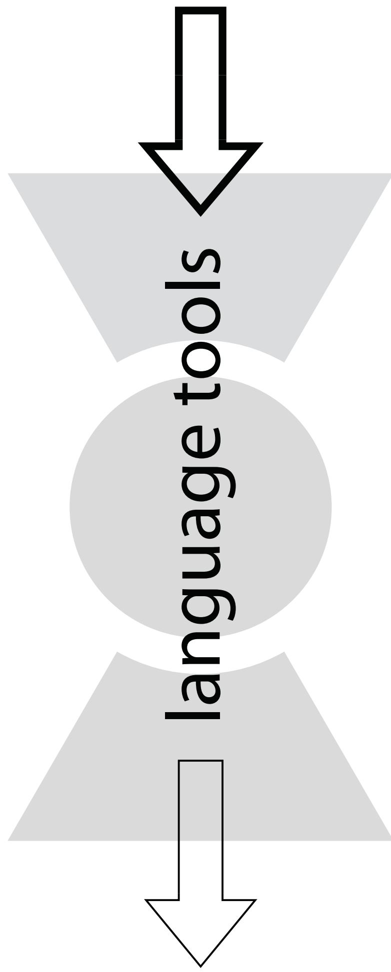
Previously on MODSOFT

Eclipse Modeling Framework



Meta-Languages

instance representation
(program, model, description)



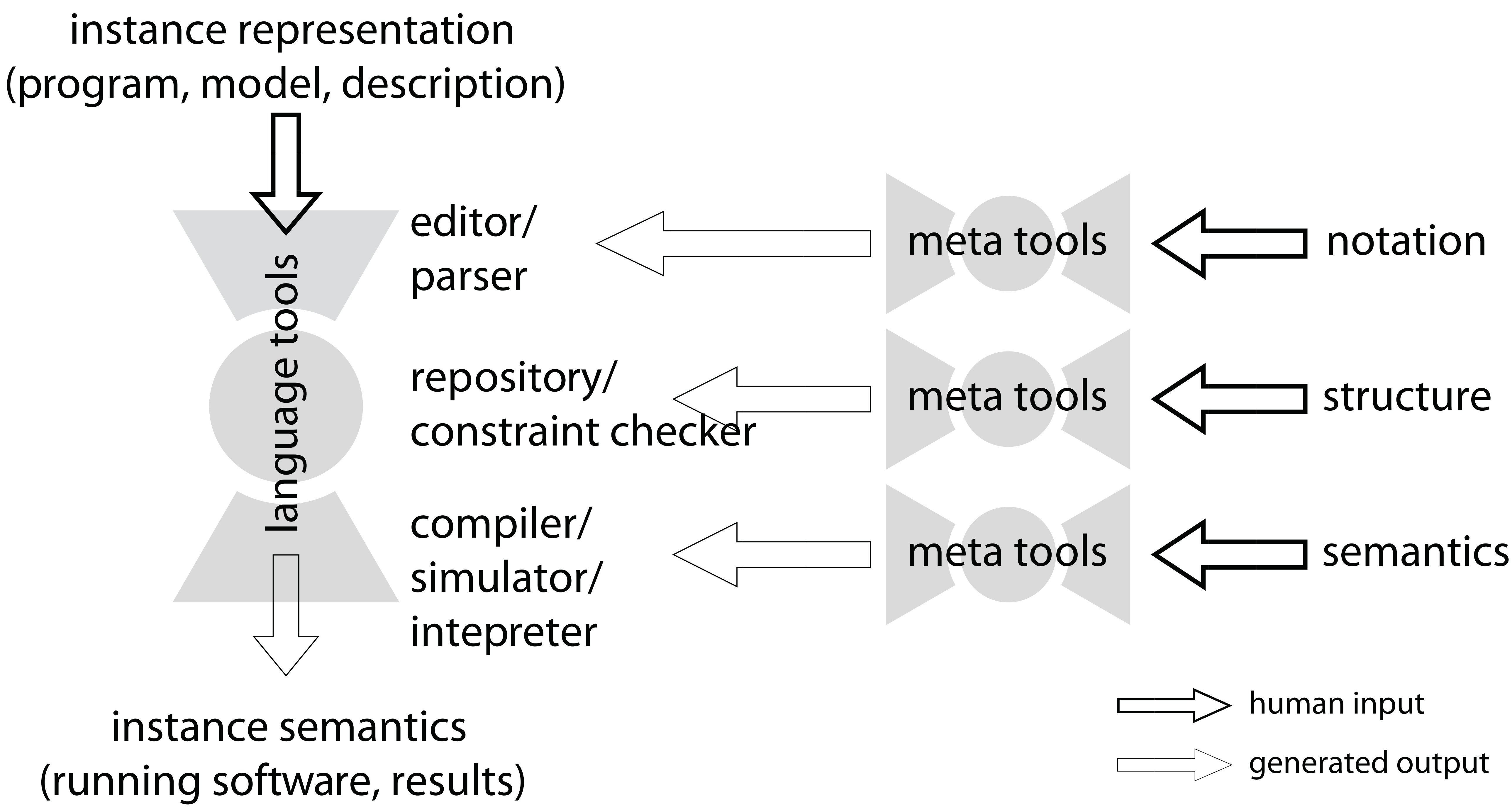
editor/
parser

repository/
constraint checker

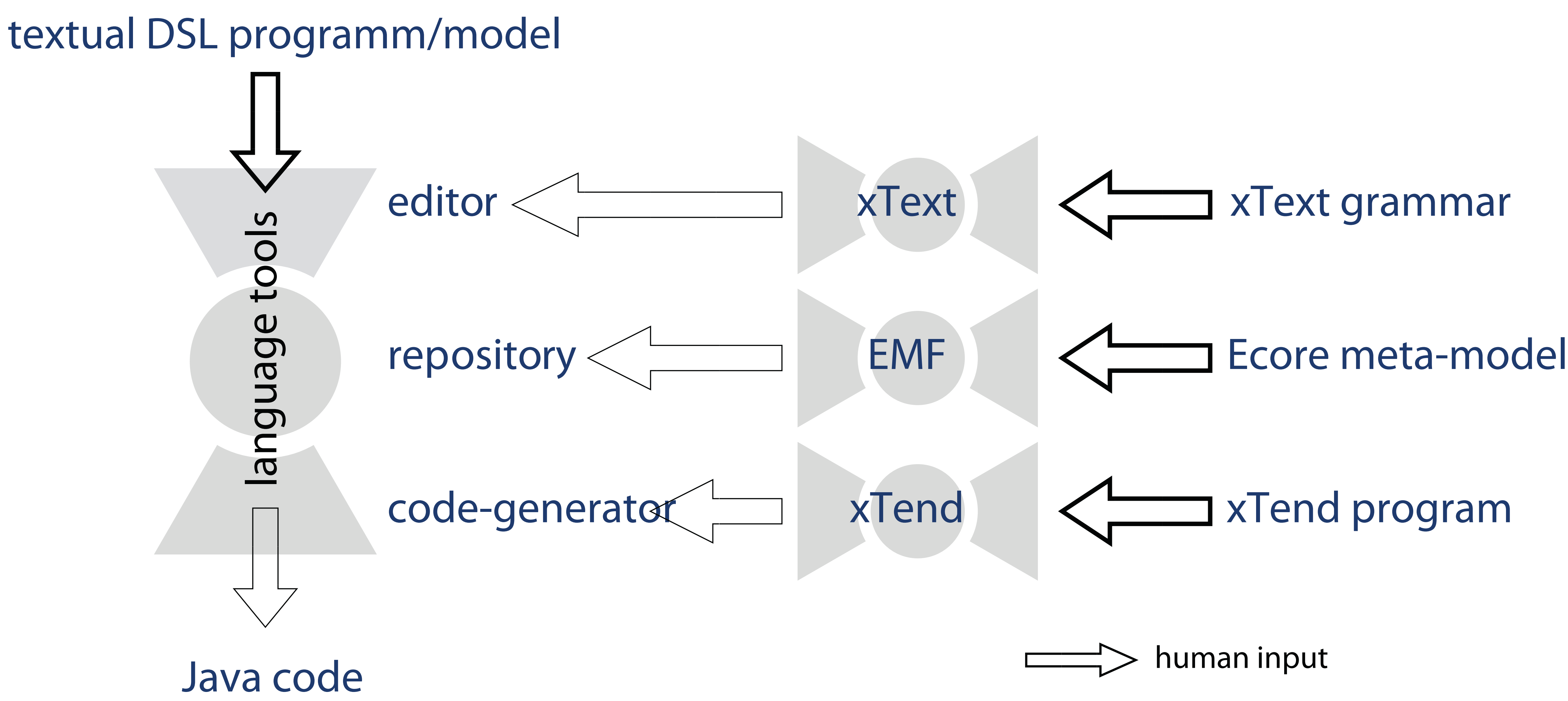
compiler/
simulator/
intepreter

instance semantics
(running software, results)

Meta-Languages



Concrete Meta-Languages



Semantics

Introduction

Different Goals of Semantics and Semantics Descriptions

- Programs
- Proving properties, calculations, simulations
- Descriptions/input for other existing systems

Different Types of Semantics

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

Operational Semantics

- The meaning of a well-formed program/model is the trace of computation steps.
- Operational semantics is also called intensional semantics, because the sequence of internal computation steps (the "intension") is most important.
- For example, two differently coded programs that both compute factorial have different operational semantics.
- Different types of operational semantics; e.g. term rewriting systems

Denotational Semantics

- The meaning of a well-formed program/model is a function from input data to output data. The steps taken to calculate the output are unimportant; it is the relation of input to output that matters.
- Denotational semantics is also called extensional semantics, because only the "extension"—the visible relation between input and output—matters.
- Thus, two differently coded versions of factorial have the same denotational semantics.
- The assignment of meaning to programs is performed compositionally.

Axiomatic Semantics

- A meaning of a well-formed program/model is a logical proposition (a "specification") that states some property about the input and output.
- Strong ties to denotational semantics, e.g. predicate logic denotations of programs/models.

Translational Semantics

- The meaning of a well-formed program/model is established via translation into a program/model of a different language with known semantics.

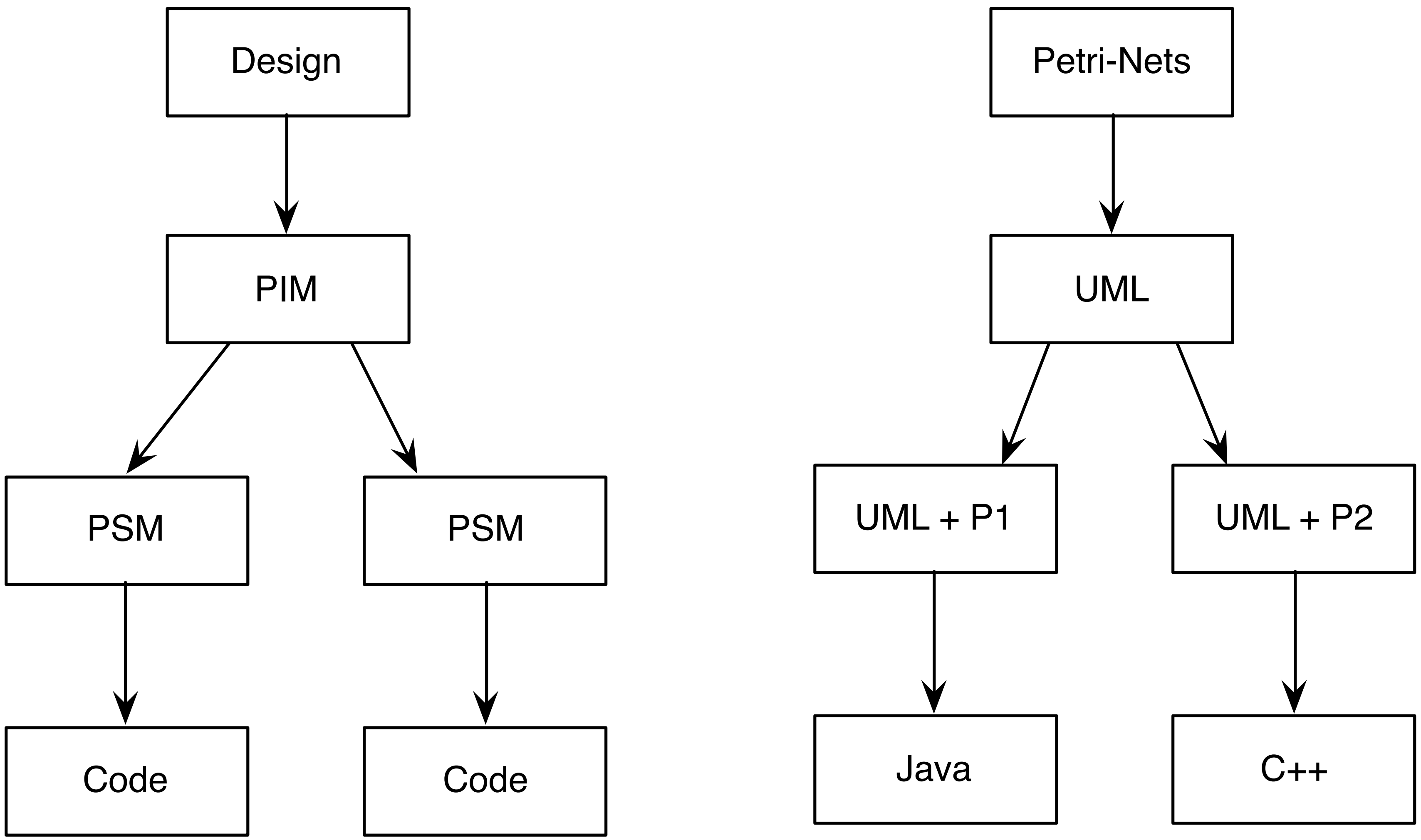
Different Types of Semantics

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

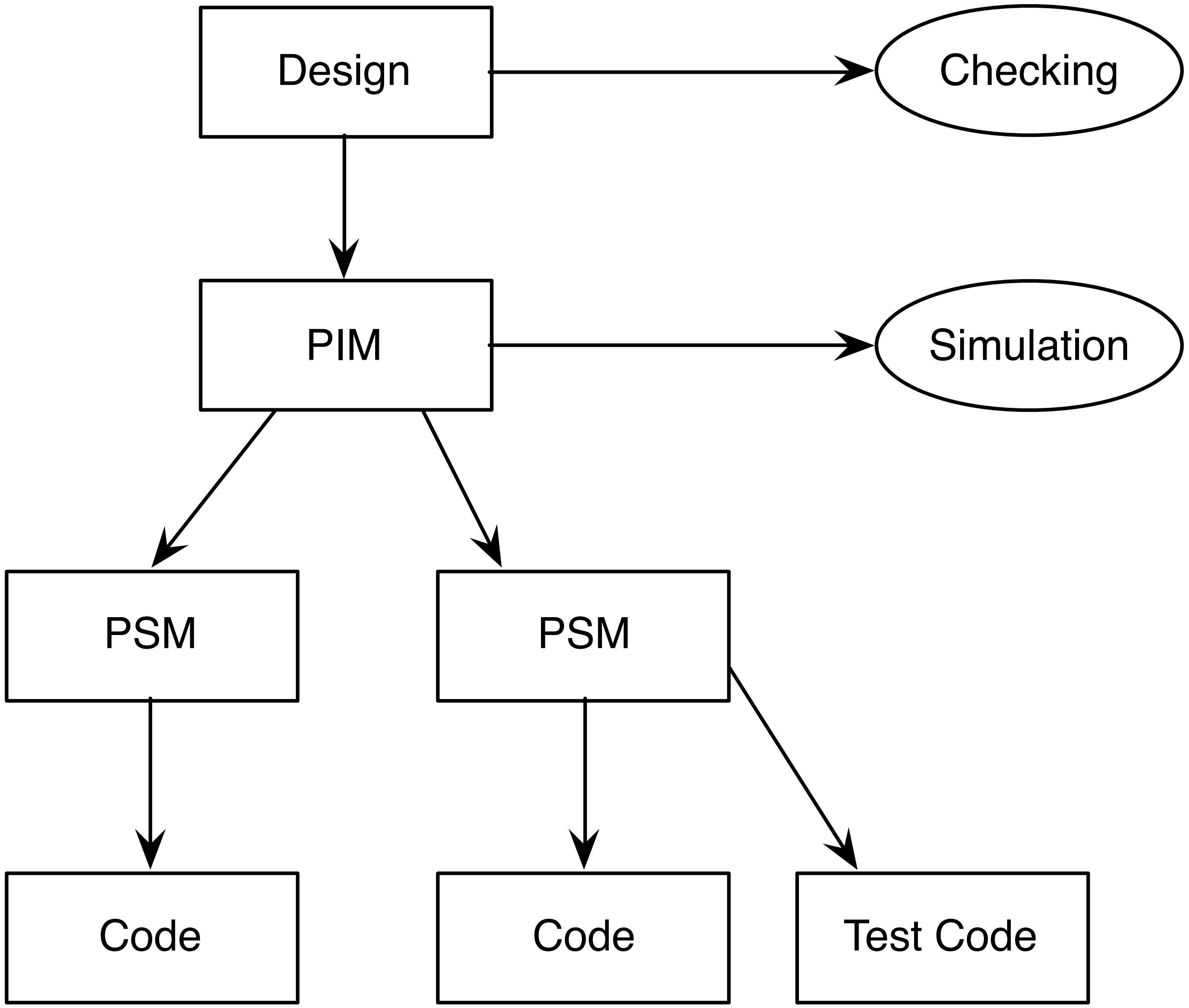
Semantics Descriptions

- Artifacts that describe how language instances (models/ programs) are mapped to a semantic domain.
- Semantics description written in a computer language to derive tools:
  - interpreter/simulator/debugger
  - compiler
  - code-generator
  - model transformations
- Different semantics descriptions languages for different types of semantics and semantic domains, e.g. M3Actions, Xtend, ATL

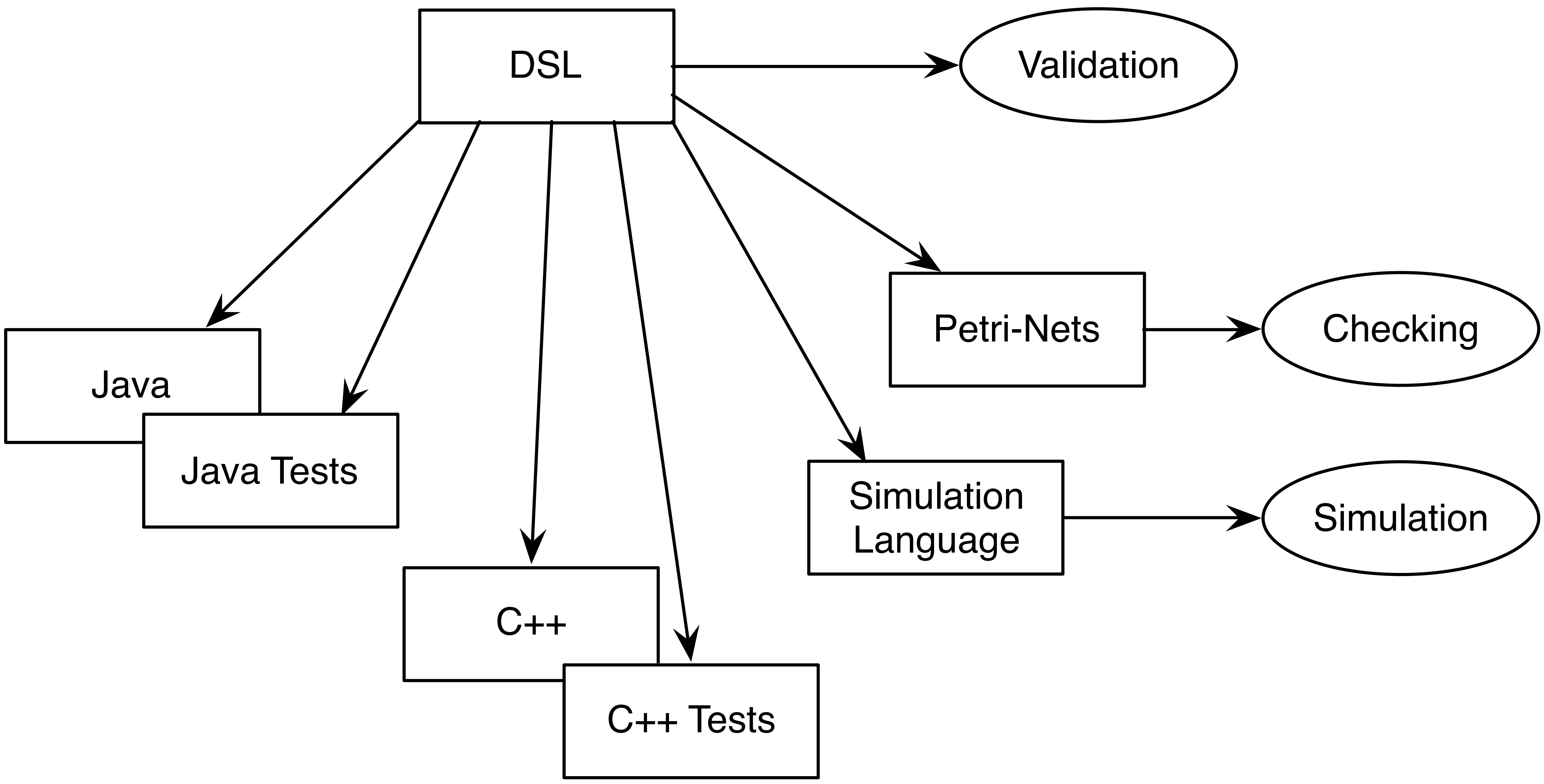
Translational Semantics and MDA



Semantics and MDA



Semantics and DSLs



Higher Order-Functions

Higher Order-Functions

- Functions/operations/methods that take other functions/... as arguments
- `List<T>.sort(comparator:(T,T)=>int): void`
- `List<T>.forAll(predicate:(T)=>boolean): boolean`
- OCL?

Mathematical Foundations

- Imperative/Procedural Programming ➡ Turing Machine
- higher-order functions/functional programming ➡ Lambda Calculus

Lambda Calculus

- Syntax

$$
\begin{aligned}
term = {} & variable \\
& \mid term\ term \\
& \mid (term) \\
& \mid \lambda variable.term
\end{aligned}
$$

- Semantics
  - alpha reduction (renaming): $\lambda y.M \Rightarrow_\alpha \lambda v.(M[y \mapsto v])$
  - beta reduction (substitution): $(\lambda x.M)N \Rightarrow_\beta M[x \mapsto N]$
- Sugar: $\lambda x.(\lambda y.yx) \equiv \lambda xy.yx$

Lambda Calculus Examples

$$T \equiv \lambda xy.x$$

$$F \equiv \lambda xy.y$$

$$if \equiv \lambda cte.cte$$

$$\begin{aligned} if\ T\ M\ N &\equiv (\lambda cte.cte)(\lambda xy.x)\ M\ N \\ &\Rightarrow (\lambda xy.x)\ M\ N \\ &\Rightarrow M \end{aligned}$$

$$and \equiv \lambda xy.(if\ x\ y\ F)$$

$$or \equiv \lambda xy.(if\ x\ T\ y)$$

Higher-Order Functions in Functional Languages

- Direct descendants form the lambda calculus
- e.g. Lisp, Scheme
- $(\lambda x.M)\ N$ ➡ `(lambda(x)M)N`

Higher-Order Functions in Imperative Languages

- Most formal properties of lambda calculus are lost, especially side-effect free (functional programming) nature of methods and operations (functions)
- Different (but similar) syntax in most languages, statically type safe
  - C#, Groovy, Scala, Java 8
- Can be realized with regular methods, polymorphy and inheritance in most object-oriented languages
- Can be realized with function pointers (and no type safety) in languages like C

Higher Order-Functions in Java

- Interfaces to declare function types
- Anonymous classes to define actual functions
- e.g. in apache commons collections or google guava

```java
public interface Predicate<T> {
    boolean apply(T value);
}

public class Collections {
    public static <T> boolean forAll(Iterable<T> iter, Predicate<T> predicate) {
        for(T value: iter) {
            if (!predicate.apply(value)) {
                return false;
            }
        }
        return true;
    }
}

public class areAllLowerCase {
    public static void main(String args[]) {
        System.out.println(
            Collections.forAll(Arrays.asList(args), new Predicate<String>() {
                public boolean apply(String value) {
                    return value.toLowerCase().equals(value);
                }
            })
        );
    }
}
```

Java 8

Java 8

```
public class Collections {
    public static <T> boolean forAll(Iterable<T> iter, Function<T, boolean> predicate) {
        for(T value: iter) {
            if (!predicate.apply(value)) {
                return false;
            }
        }
        return true;
    }
}

public class areAllLowerCase {
    public static void main(String args[]) {
        System.out.println(
            Collections.forAll(
                Arrays.asList(args),
                (String v) -> v.toLowerCase().equals(v));
        );
    }
}
```

Scala

```
def forAll(list:List[T], predicate: T => Boolean) {
    for (value <- list) {
        if (!predicate(value)) {
            return false;
        }
    }
    return true;
}

println(forAll(Arrays.asList(args), v => v.toLowerCase().equals(v)));
```

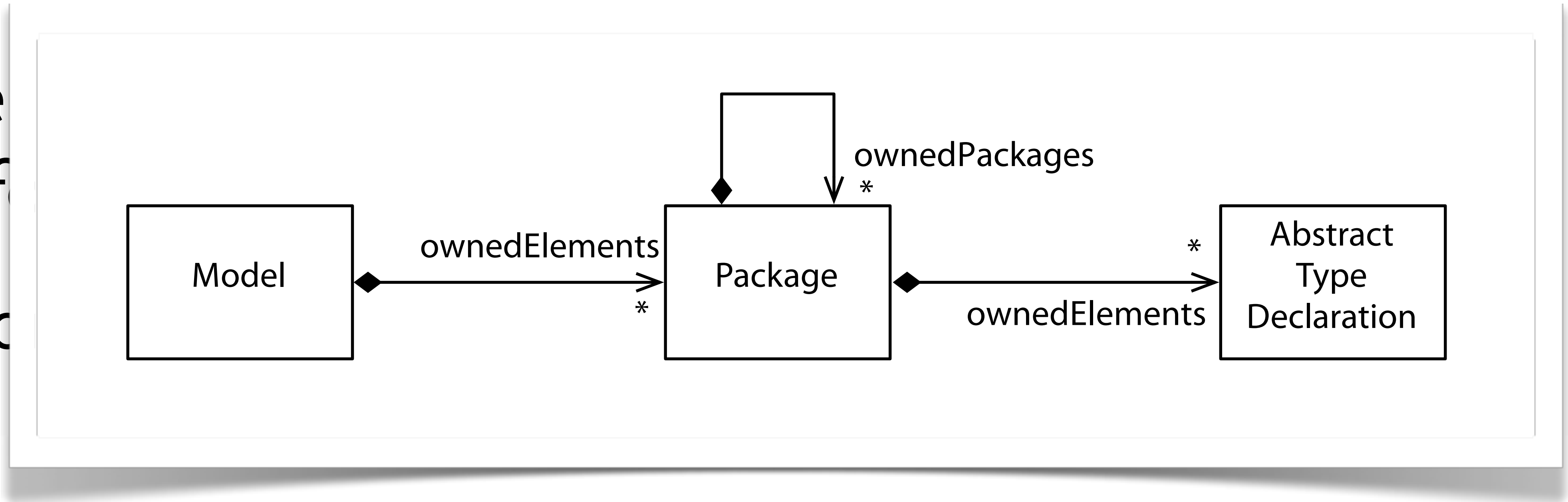
OCL-like Internal DSL in Scala

- *OCL-like* internal Scala DSL analog to our internal Scala model transformation language [1]
- OCL collection operations mapped to Scala's higher-order fuctions [2]:

1. **L. George, A. Wider, M. Scheidgen:** *Type-Safe Model Transformation Languages as Internal DSLs in Scala*; Theory and Practice of Model Transformations - 5th International Conference, ICMT; 2012
2. **Filip Krikava**: *Enrichting EMF Models with Scala;* Slideshare

OCL-like Internal DSL in Scala

- *OCL-like* inte
  model transf
- OCL collectic
  fuctions [2]:



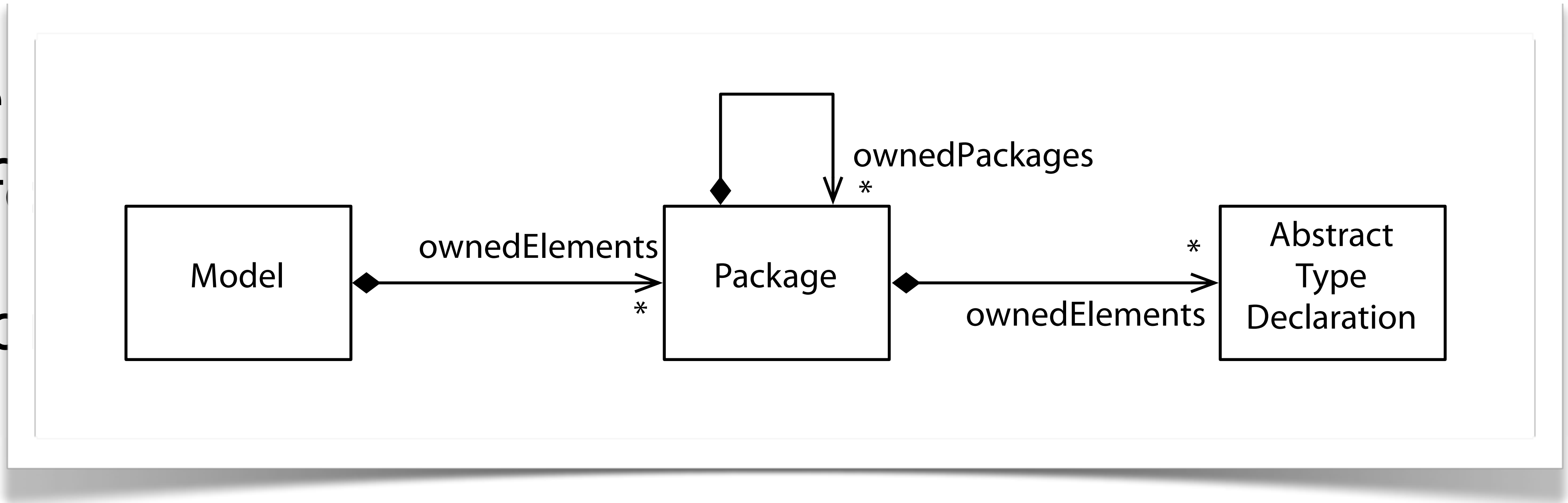
pure OCL

```
context Model:
    self.ownedElements->collect(p|p.ownedElements)->size
```

1. **L. George, A. Wider, M. Scheidgen:** *Type-Safe Model Transformation Languages as Internal DSLs in Scala*; Theory and Practice of Model Transformations - 5th International Conference, ICMT; 2012
2. **Filip Krikava**: *Enrichting EMF Models with Scala;* Slideshare

OCL-like Internal DSL in Scala

- *OCL-like* inte
  model transf
- OCL collecti
  fuctions [2]:



pure OCL

```
context Model:
    self.ownedElements->collect(p|p.ownedElements)->size
```

OCL-like expression in Scala

```
def numberOfFirstPackageLevelTypes(self: Model): Int =
    self.getOwnedElements().collect(p=>p.getOwnedElements()).size()
```


1. **L. George, A. Wider, M. Scheidgen:** *Type-Safe Model Transformation Languages as Internal DSLs in Scala*; Theory and Practice of Model Transformations - 5th International Conference, ICMT; 2012
2. **Filip Krikava**: *Enrichting EMF Models with Scala;* Slideshare

With great power comes great responsibility

- Code that uses higher-order functions can be very very hard to read and maintain
- Higher-order functions that are used according to certain well known patterns can be very readable and still be powerful
  - filters, predicate logical functions, transformations, etc. for complex datatype classes like collections
  - callbacks, event handlers, etc.
- Avoid anonymous constructs: verbose code has its merits, names function as build in documentation
- There is a reason why complex, robust, and commercially successful software software is not written in Lisp, Scheme, or Haskel. *(Please don't say emacs)*

Higher-Order Functions and Semantics

- Higher-Order functions for list operations greatly improves the ability to navigate and query EMF-models
- Most languages that require model navigation/queries have higher-order function constructs, e.g. collections in OCL

Operational Semantics

with EMF

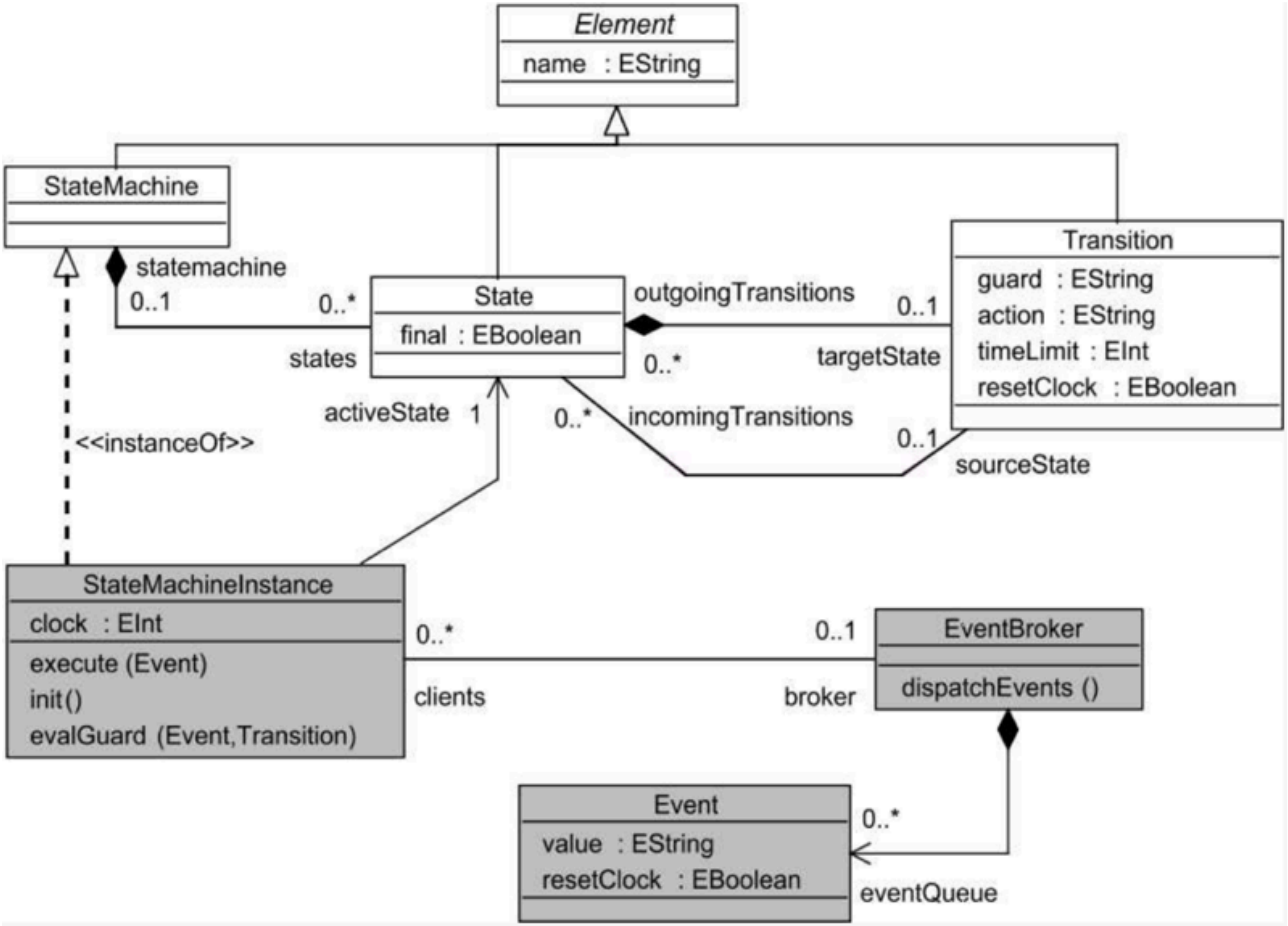
Approaches to Operational Semantics with EMF

- 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*

Abstract Syntax and Runtime Concepts



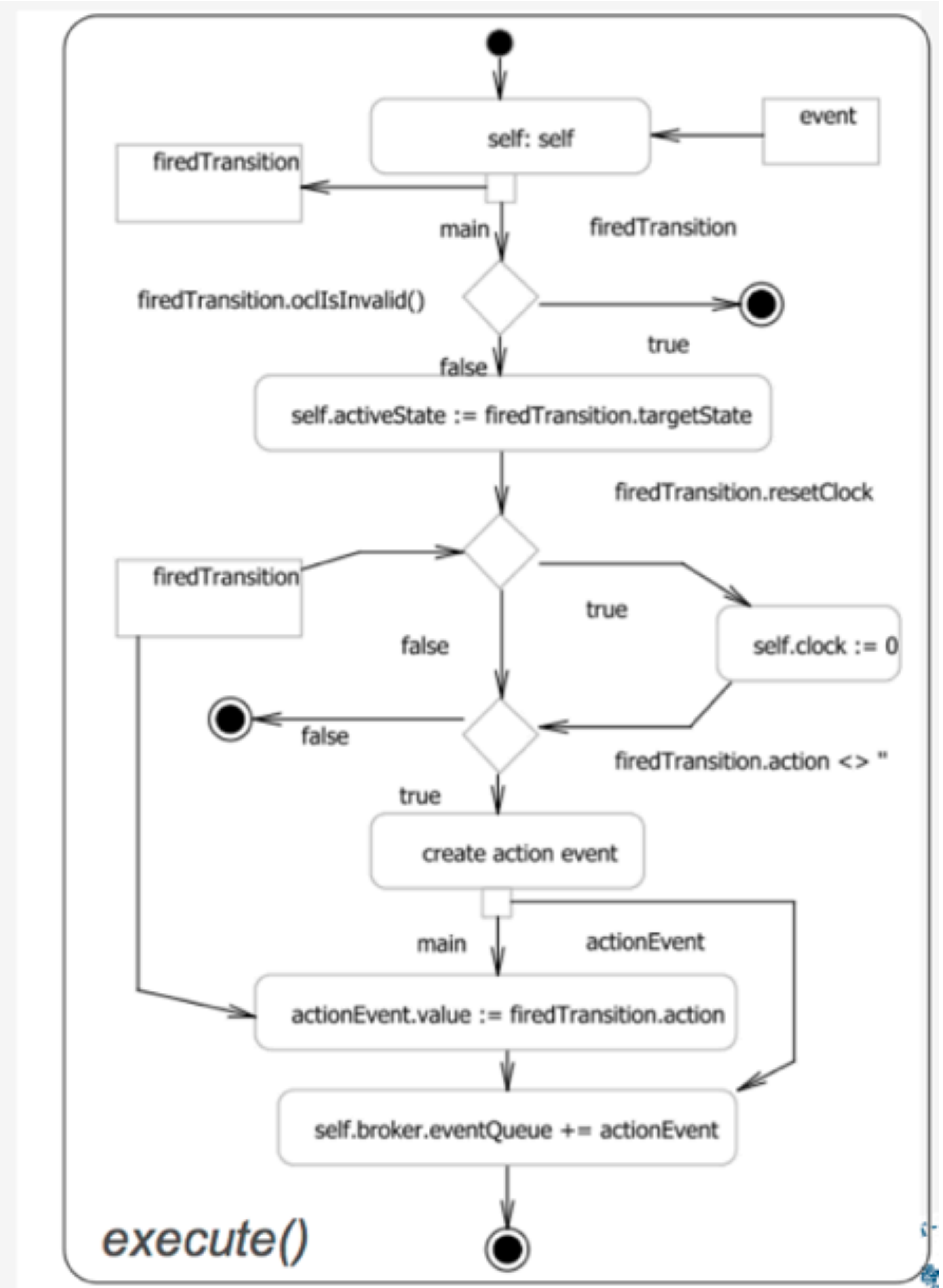
Meta-Model Operations and Operational Semantics

- 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. *M3Actions*
  - 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

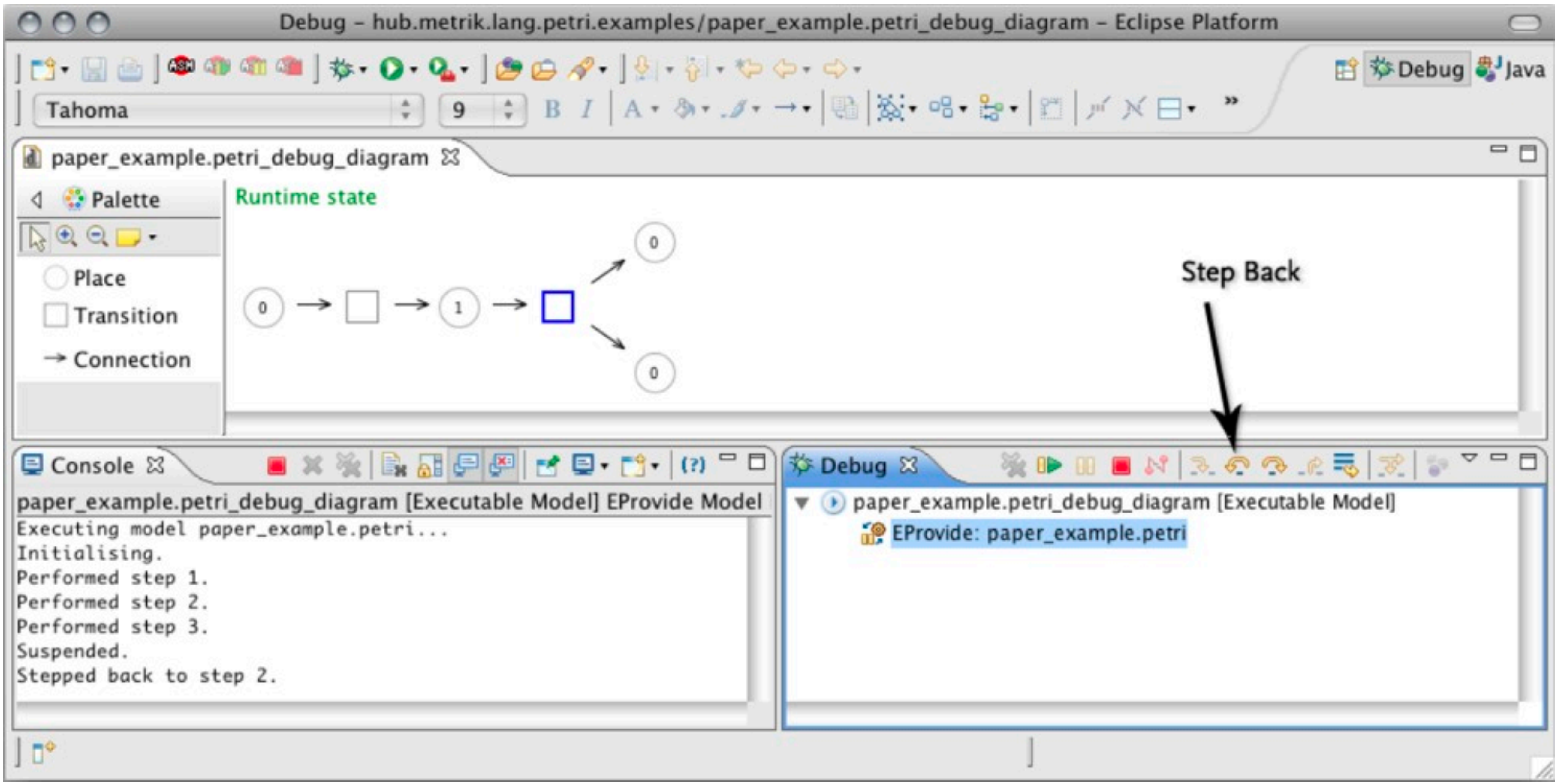
Example Semantics with Actions (MActions)



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

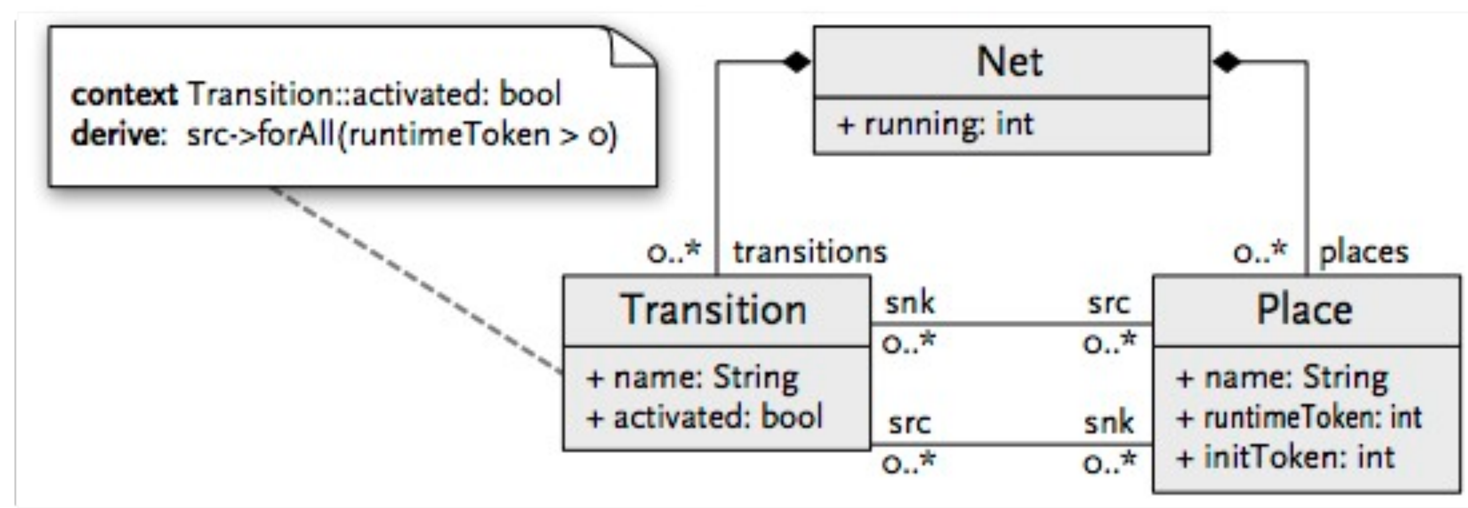


Traces and Debugging

```
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
    }
    // ...
}
```



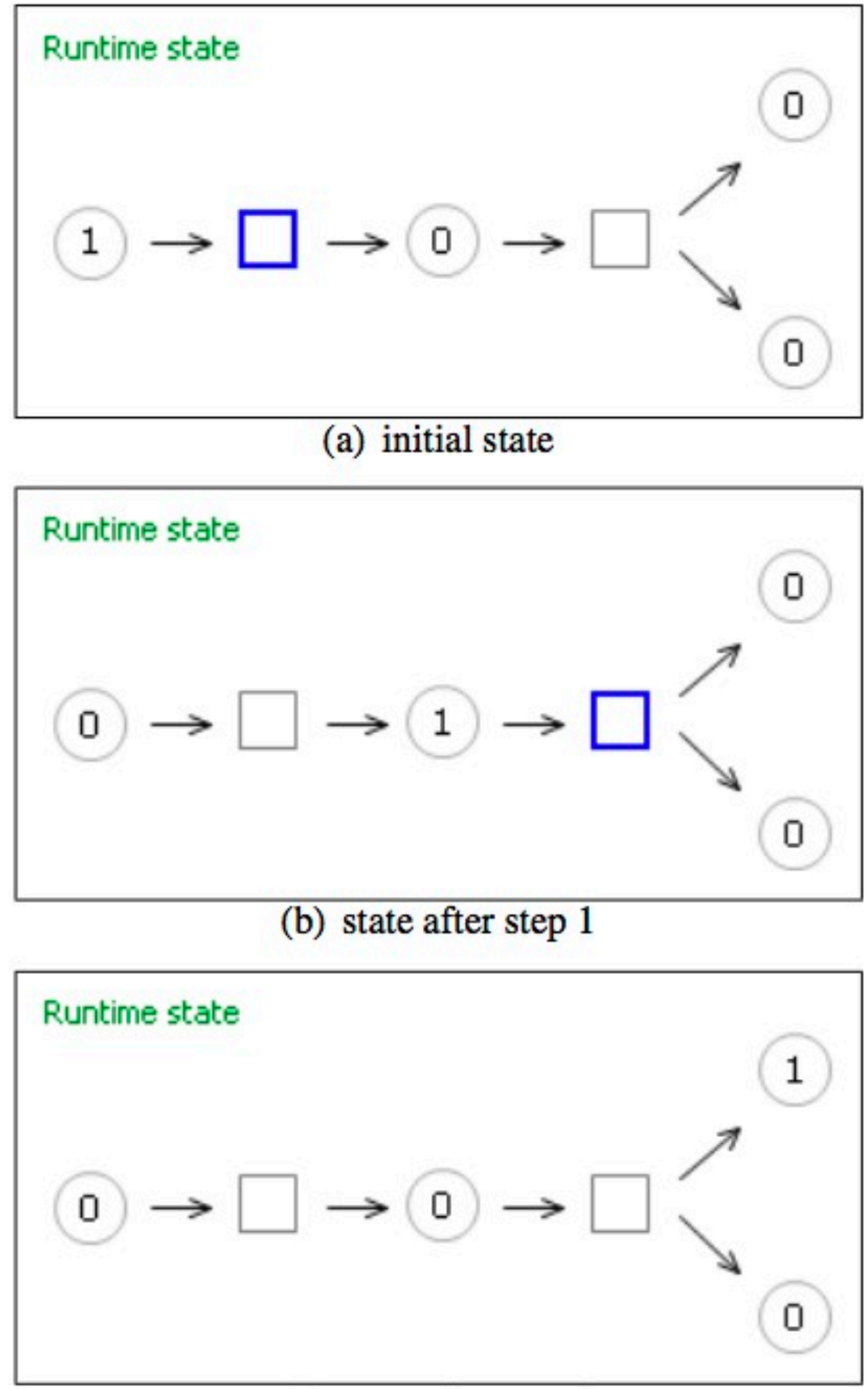
Traces and De

```java
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
    }
    // ...
}
```

context Transiti
derive: src->for



(a) initial state

(b) state after step 1

(c) state after step 2

Traces and De

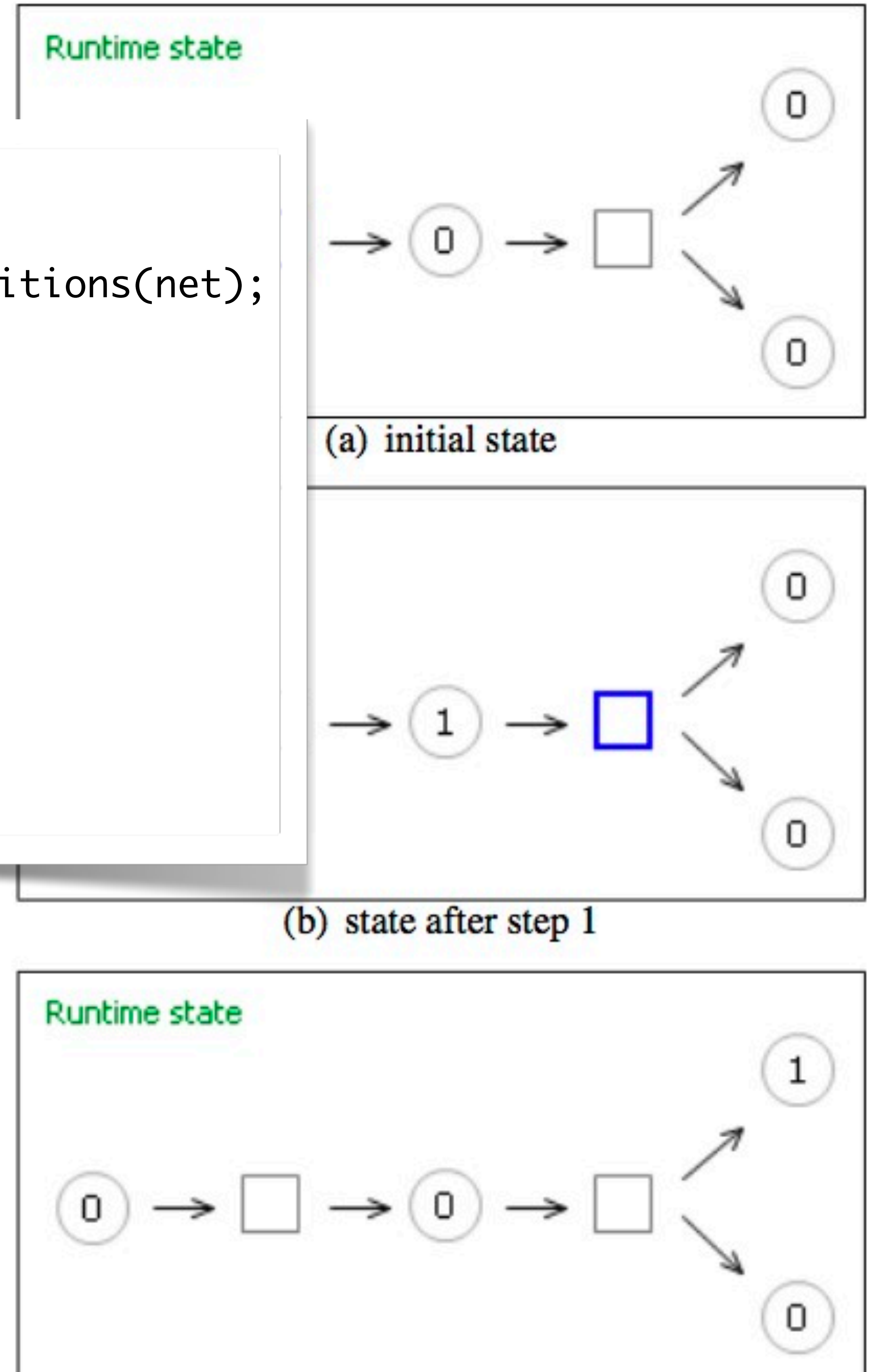
```
    // ...
    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransitions(net);
        if (!ats.isEmpty()) {
            Transition t = choose(ats);
            for (Place p : t.getSrc())
                consume(p);
            for (Place p : t.getSnk())
 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

    // ...
```

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

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

context Transiti
derive: src->for



(a) initial state

(b) state after step 1

(c) state after step 2

Traces and De

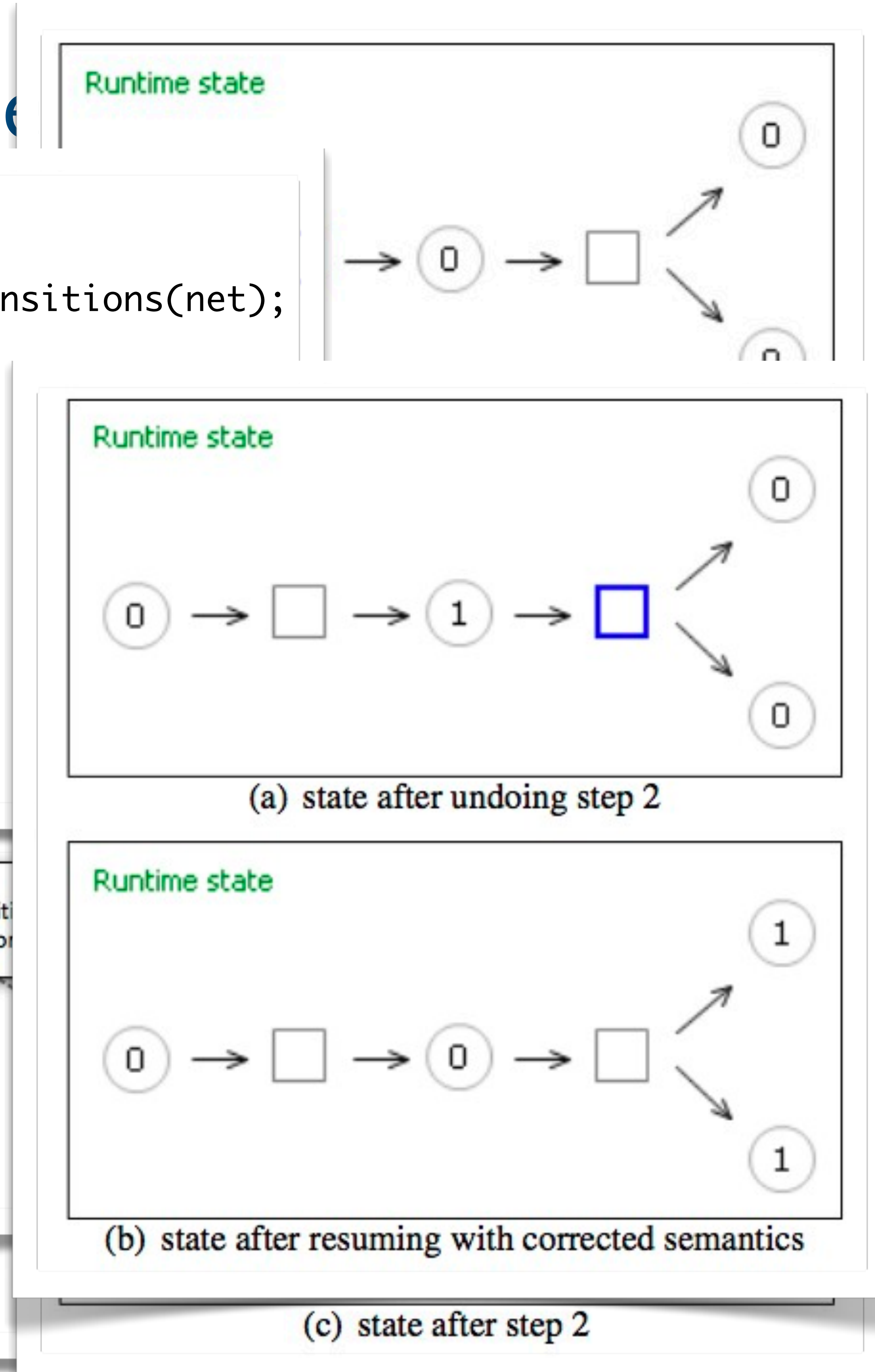
```java
    // ...
    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransitions(net);
        if (!ats.isEmpty()) {
            Transition t = choose(ats);
            for (Place p : t.getSrc())
                consume(p);
            for (Place p : t.getSnk())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

    // ...
```

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

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

context Transiti
derive: src->fo

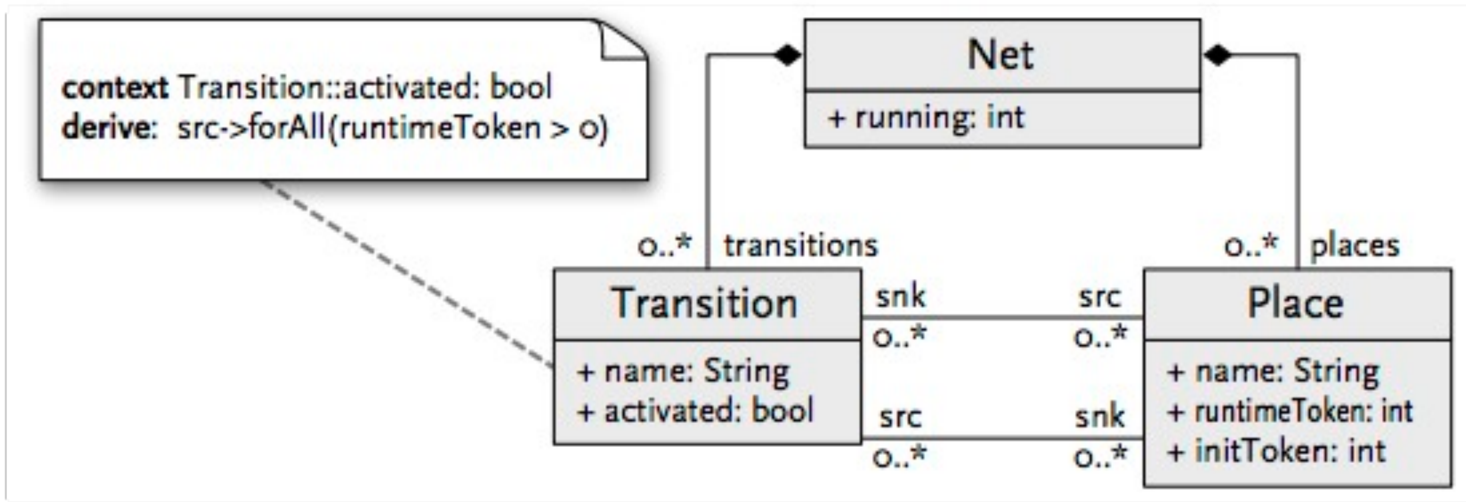


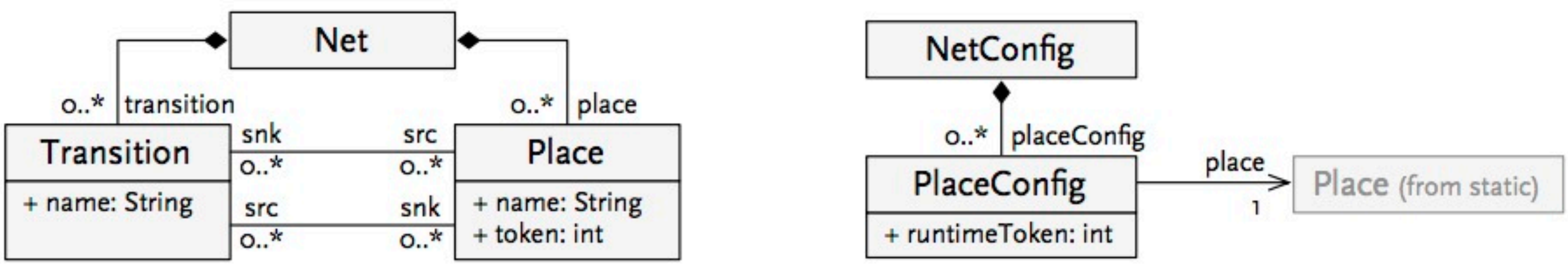
(a) state after undoing step 2

(b) state after resuming with corrected semantics

(c) state after step 2

Traces and Debugging





(a) Static metamodel.

(b) Separate configuration metamodel.

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

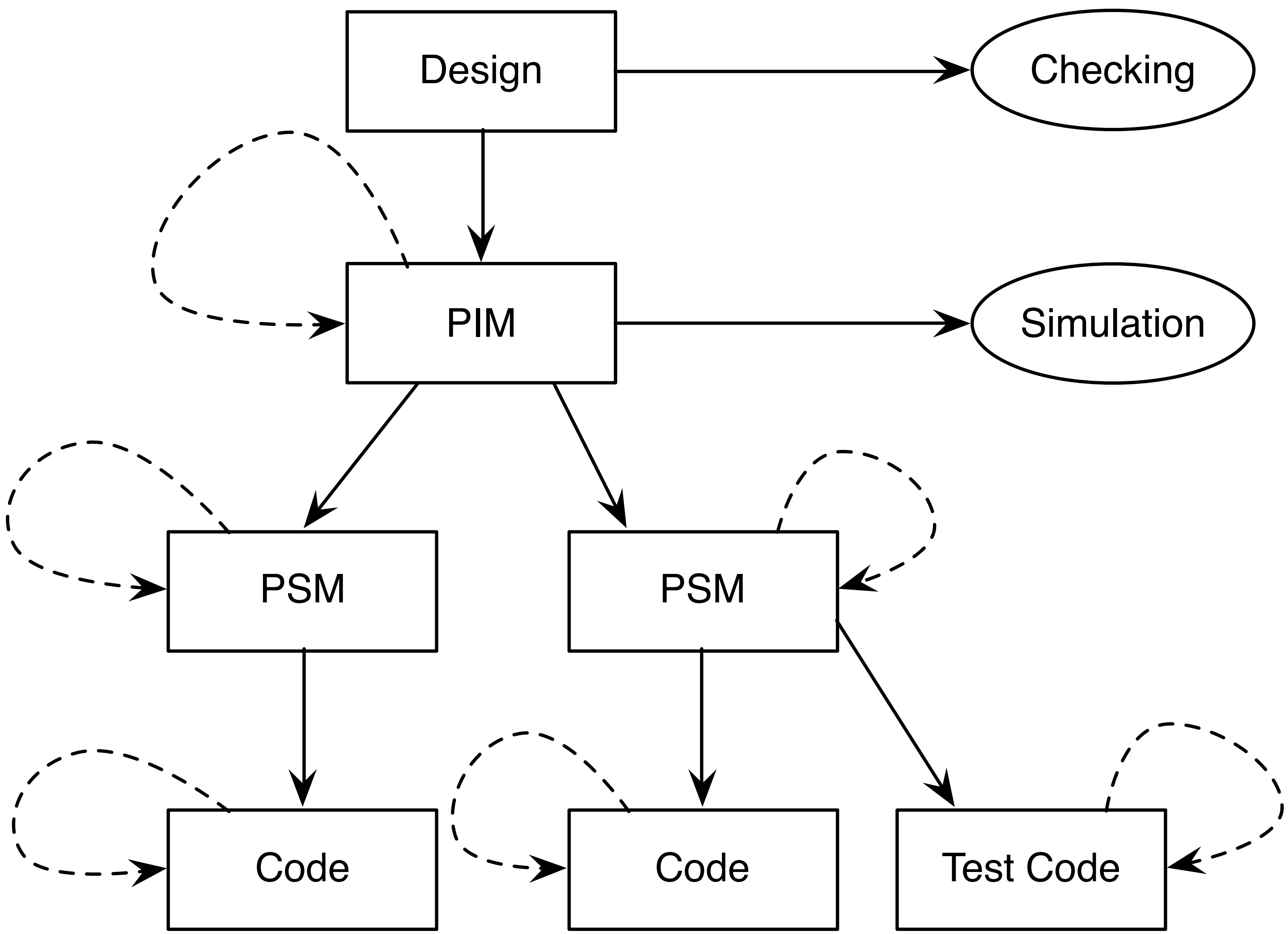
Types of "Model Transformations"

- 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