Behavioral Semantics of Languages

Julien Deantoni

Modéliser un langage

- Une syntaxe abstraite (et concrète) décrivant un langage et permettant de définir des comportements (des modèles)
- Une sémantique expliquant comment les programmes conformes à la grammaire (les comportements) sont executés

syntax	*semantics*
while (*b*) do C ; done	Exécuter C de manière répétée (et sequentielle), aussi longtemps que l'expression *b* est vraie.

Modéliser un langage

- Une syntaxe abstraite (et concrète) décrivant un langage et permettant de définir des comportements (des modèles)
- Une sémantique expliquant comment les programmes conformes à la grammaire (les comportements) sont executés

syntax

```
while (b)
do
    C ;
done
```

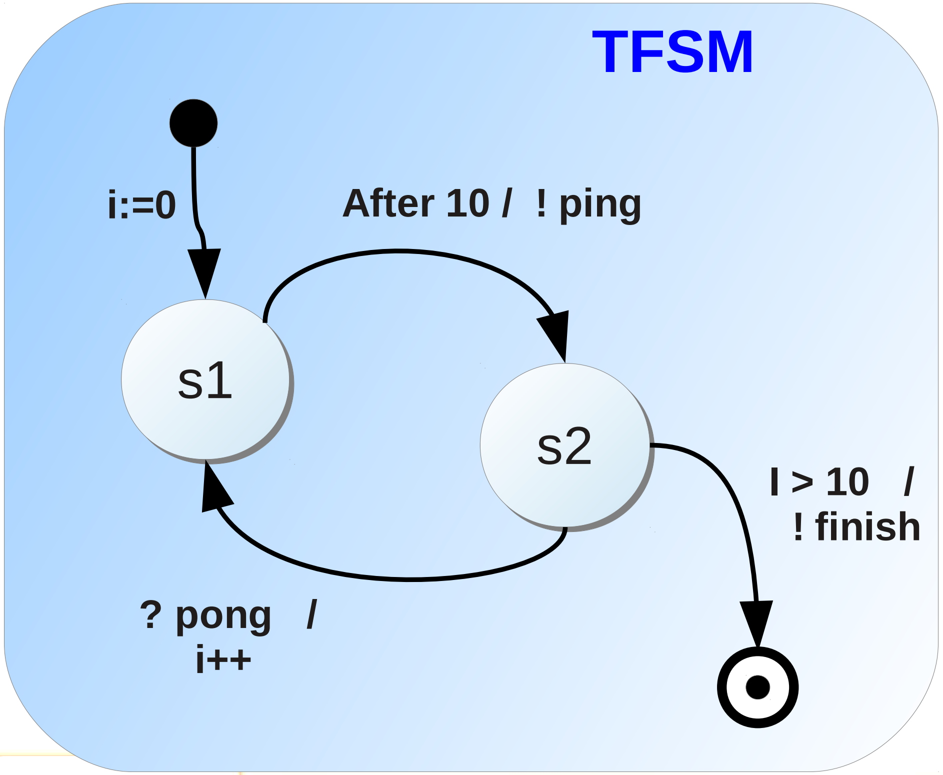
semantics

Exécuter C de manière répétée (et sequentielle), aussi longtemps que l'expression *b* est vraie.

- 1) évaluer l'expression *b.*
  - si *b* == vrai, exécuter C et retourner à 1)
  - si *b* == faux, sortir.

Modéliser un langage

- Une syntaxe abstraite (et concrète) décrivant un langage et permettant de définir des comportements (des modèles)
- Une sémantique expliquant comment les programmes conformes à la grammaire (les comportements) sont executés



Nous avons ici plusieurs sémantiques possibles

Why modeling behavioral semantics

- People learning the language can understand the subtleties of its use
- The model over which the semantics is defined (the semantic domain) can indicate what the requirements are for implementing the language (as a compiler/interpreter/…)
- Global properties of any program written in the language, and any state occurring in such a program, can be understood from the formal semantics
- Implementers of tools for the language (parsers, compilers, interpreters, debuggers etc) have a formal reference for their tool and a formal definition of its correctness/completeness
- Programs written in the language can be verified formally against a formal specification (or at least a definition for their correctness exists)
- 2 different programs in the language can be proved formally as equivalent/non-equivalent
- **From a computer readable version of the semantics, an interpreter can be automatically generated – full compiler generation is not (yet) feasible**

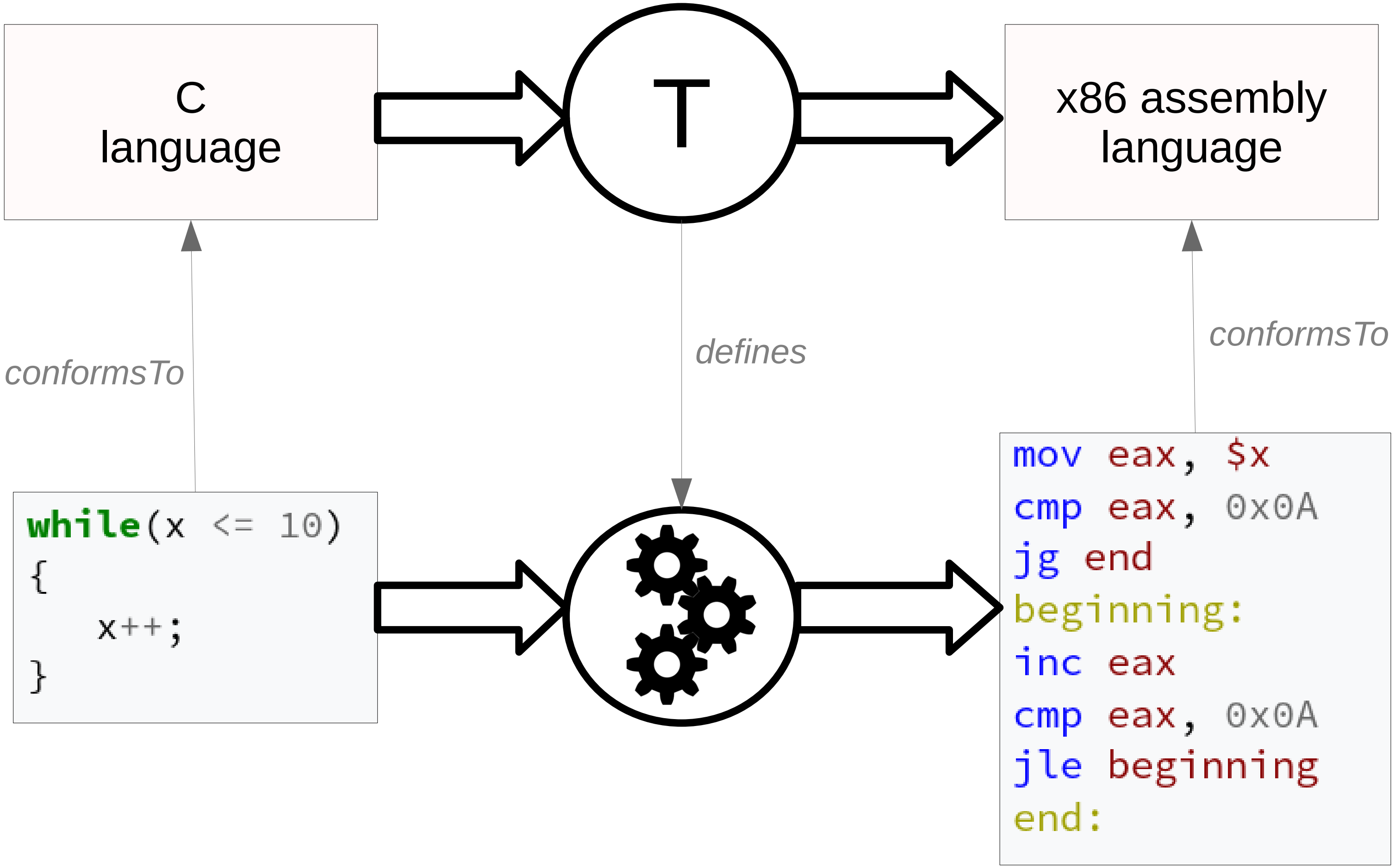
http://www.cs.tau.ac.il/~msagiv/courses/pa07/Operational_Semantics.pdf

Sémantique comportementale

- Les principales manières de décrire la sémantique :
  - **Transformational :** the semantics is defined by reducing constructs of the language to more elementary ones by means of definitional transformations into a simpler language whose the semantics is already given.
    - **Denotational :** the meaning of the program is given by the translation of a program to a mathematical function, which maps the state of the machine before execution to the state after execution.
  - **Axiomatic :** the semantics is defined by a logical theory associated to each language elements in order to enable some properties to be proven (the formulae describe, for each statement, the relation between the pre-state and the post-state of the statement execution)
  - **Operational :** the meaning is given by defining an abstract interpreter of the language where rules define how operators modify the system state.
    - **Attribute Grammar :** the semantics is defined by decorations of a context free grammar with attributes you are interested in. Basically attributes can take values from arbitrary domains and arbitrary functions can be specified, written in a language of choice, to describe how attributes values in rules are derived from each other. The set of attributes defines the state of the system

Transformational semantics

- **Transformational :** the semantics is defined by reducing constructs of the language to more elementary ones by means of definitional transformations into a simpler language whose the semantics is already given.



Axiomatic semantics

- **Axiomatic :** the semantics is defined by a logical theory associated to each language elements in order to enable some properties to be proven (the formulae describe, for each statement, the relation between the pre-state and the post-state of the executing the statement)

Hoare Triples

- Meaning of construct S can be described in terms of triples:

  { P } S { Q }

  - P and Q are formulas or assertions.
    - P is a precondition on S
    - Q is a postcondition on S
  - Asserts a fact (may be either true or false)
  - The triple is valid if:
    - execution of S begins in a state satisfying P
    - S terminates
    - resulting state satisfies Q

http://www.cs.purdue.edu/homes/suresh/565-Spring2009/lectures/lecture-6.pdf

Axiomatic semantics

- **Axiomatic :** the semantics is defined by a logical theory associated to each language elements in order to enable some properties to be proven (the formulae describe, for each statement, the relation between the pre-state and the post-state of the executing the statement)

```
while (b)
do
    C ;
done
```

{ ? } **while** B **do** C **od** { P }

{ P } **while** B **do** C **od** { P }

{ P } **while** B **do** C **od** { P and not B }

```
while(x <= 10)
{
    x++;
}
```

{x ∈ ℤ} **while** B **do** C **od** { x ∈ ℤ ∧ x > 10}

- P satisfying this rule is called a *loop invariant* because it must hold before and after the each iteration of the loop
- There is NO algorithm for computing the correct P; it requires intuition and an understanding of why the program works

https://courses.engr.illinois.edu/cs421/fa2018/CS421D/lectures/27-28-HoareLogic-2x3.pdf

Operational semantics

- The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. These sequences then are the meaning of the program.
- Structural Operational Semantics [http://homepages.inf.ed.ac.uk/gdp/publications/sos_jlap.pdf]

$$\frac{\text{Condition}}{\text{Rewriting rule}}$$

$\langle n, \sigma \rangle \Downarrow n$ "n in state σ, evaluates to n"

$\langle a, \sigma \rangle \Downarrow n$ "expression a in state σ, evaluates to n"

$\langle X, \sigma \rangle \Downarrow \sigma(X)$ "location X evaluates to its contents in a state"

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma} \quad \text{(while loops)}$$

while (*b*)
do
C ;
done

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{true} \quad \langle c, \sigma \rangle \Downarrow \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma'' \rangle \Downarrow \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \Downarrow \sigma'}$$

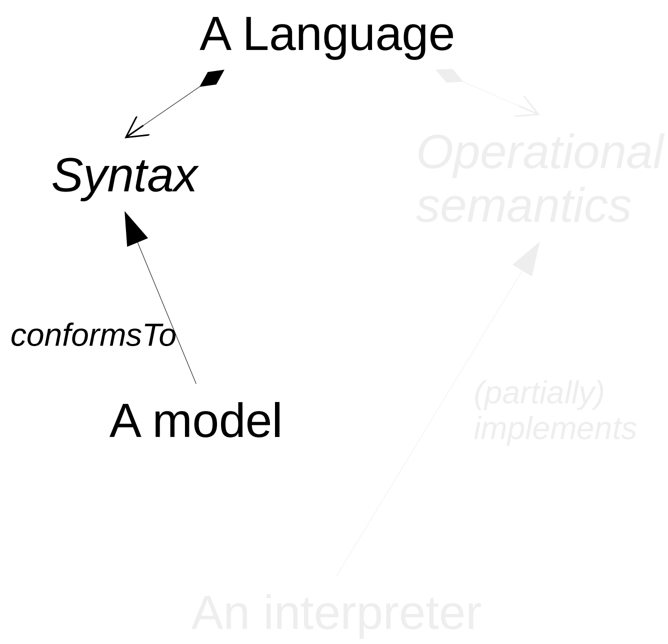
$$\frac{\langle B, s \rangle \Rightarrow \textbf{true}}{\langle \textbf{while } B \textbf{ do } C, s \rangle \longrightarrow \langle C; \textbf{while } B \textbf{ do } C, s \rangle} \qquad \frac{\langle B, s \rangle \Rightarrow \textbf{false}}{\langle \textbf{while } B \textbf{ do } C, s \rangle \longrightarrow s}$$

From wikipedia

GEMOC approach : context

- We consider models that can be interpreted according to their (concurrent and timed) operational semantics
- We do not want to implement all the tooling for each new language

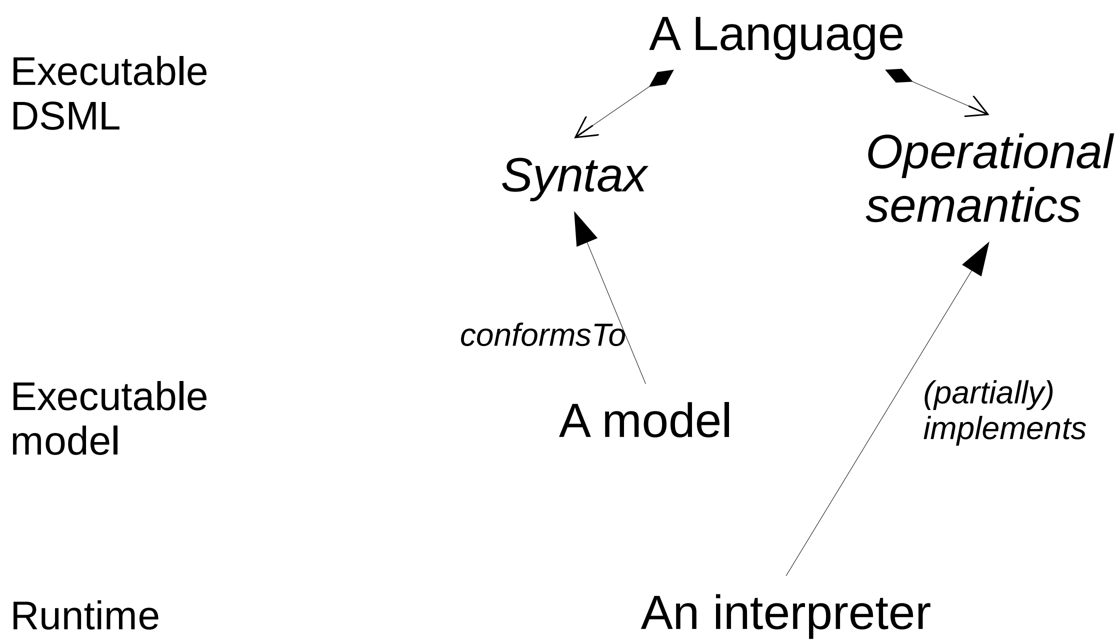
DSML



model

GEMOC approach : context

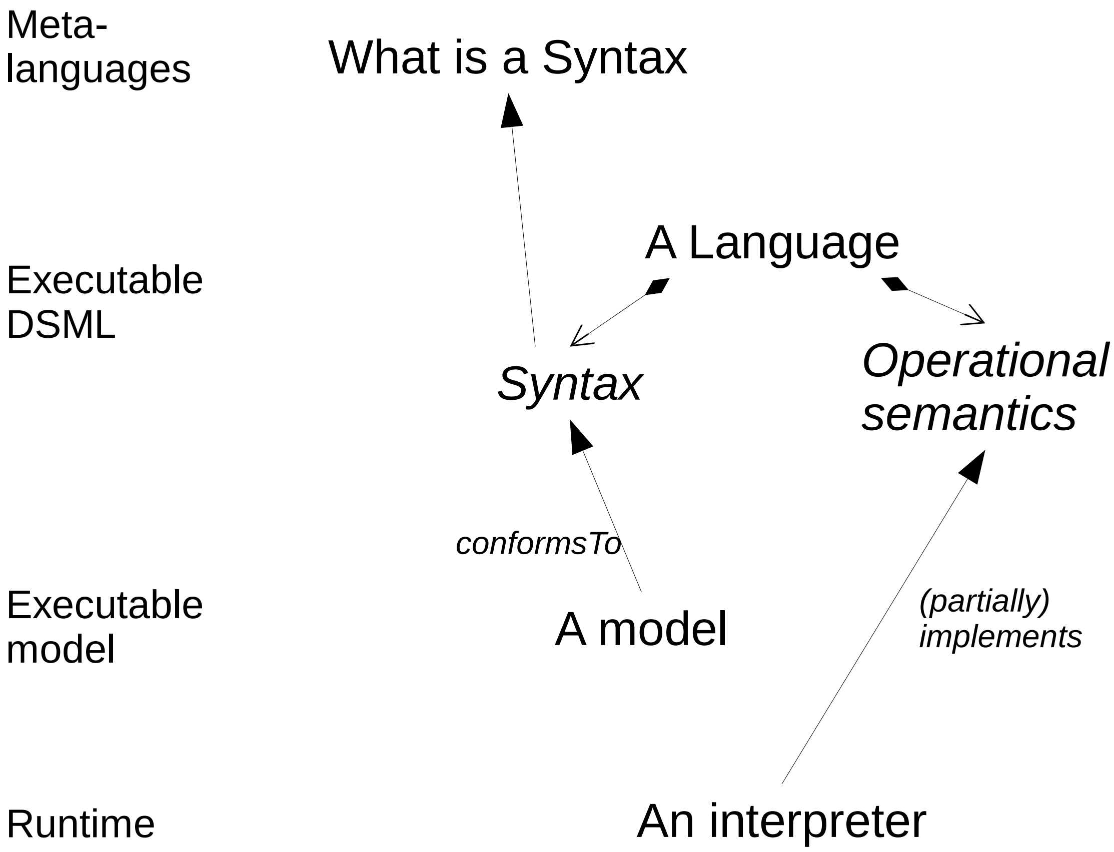
- We consider models that can be interpreted according to their (concurrent and timed) operational semantics
- We do not want to implement all the tooling for each new language



We need to make the operational semantics explicit… and as formal as possible

GEMOC approach : context

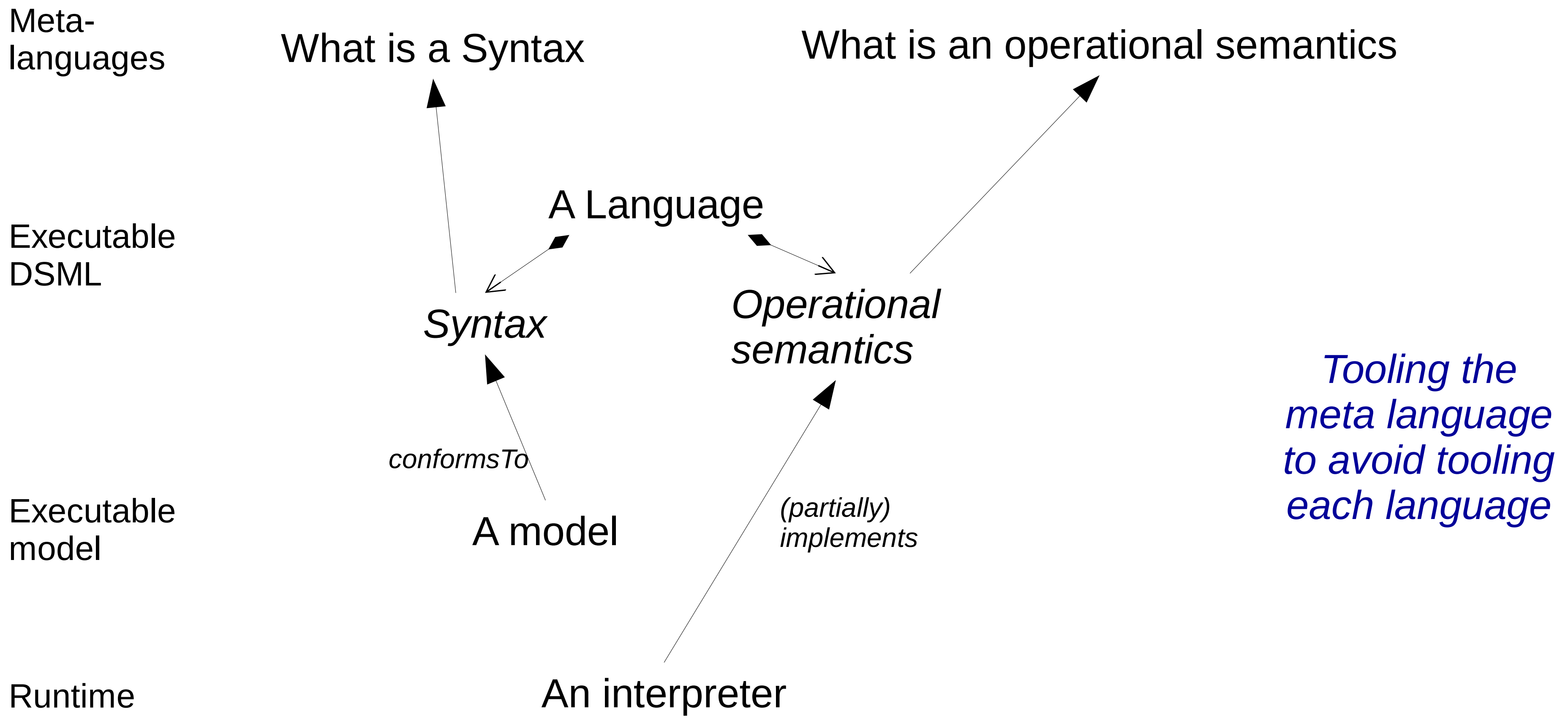
- We consider models that can be interpreted according to their (concurrent and timed) operational semantics
- We do not want to implement all the tooling for each new language



Tooling the meta language to avoid tooling each language

GEMOC approach : context

- We consider models that can be interpreted according to their (concurrent and timed) operational semantics
- We do not want to implement all the tooling for each new language



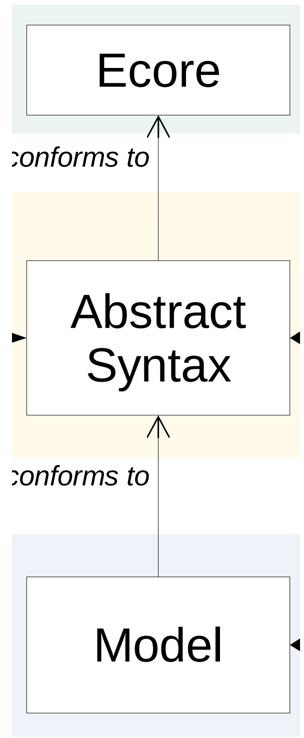
The GEMOC approach

Meta-
languages

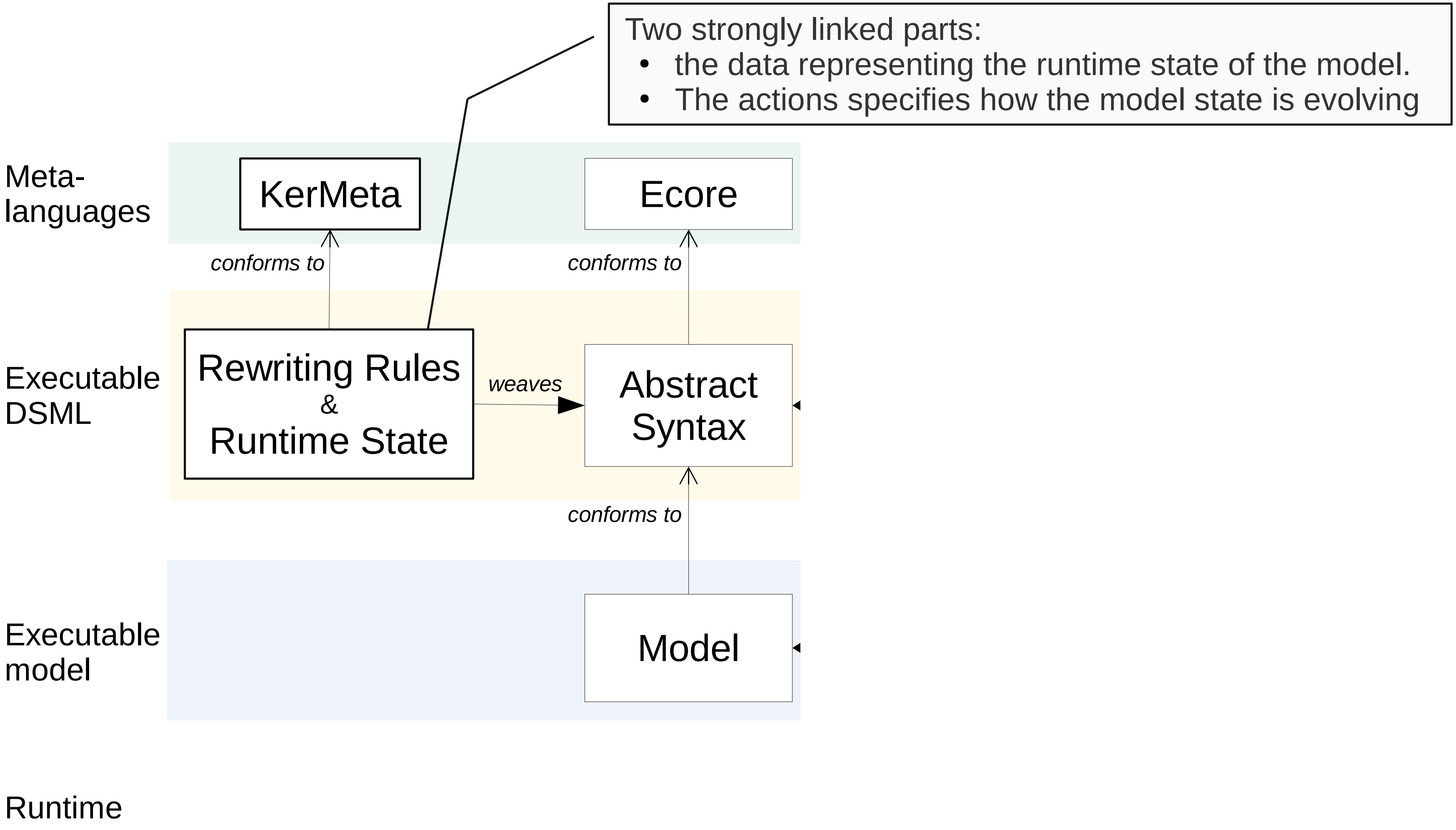
Executable
DSML

Executable
model

Runtime



The GEMOC approach



Kermeta 3 (K3)

http://diverse-project.github.io/k3/

- K3 is an action language built on top of the Xtend programming language and mainly used to implement the execution semantics of Ecore metamodels. Concretely, K3 allows to "re-open" the classes generated from an Ecore metamodel using simple annotations in order to weave new features and operations.
- Main features of K3 include:
  - Executable metamodeling: Using K3, one can insert new methods in existing Ecore meta-classes, with their implementation. These methods define the execution semantics of the corresponding metamodel in the form of an interpreter;
  - Metamodel extension: The very same mechanism can be used to extend existing Ecore metamodels and insert new features (eg. attributes) in a non-intrusive way;
  - Full Java compatibility: K3 files are plain Xtend files. As such, K3 files are ultimately compiled as plain Java code. This means that Java code and API can be used in K3 files and vice versa.
- . We can use it to weave the **state** and the **rewriting rules**, *e.g.,*

From wikipedia

Kermeta 3 (K3)

http://diverse-project.github.io/k3/

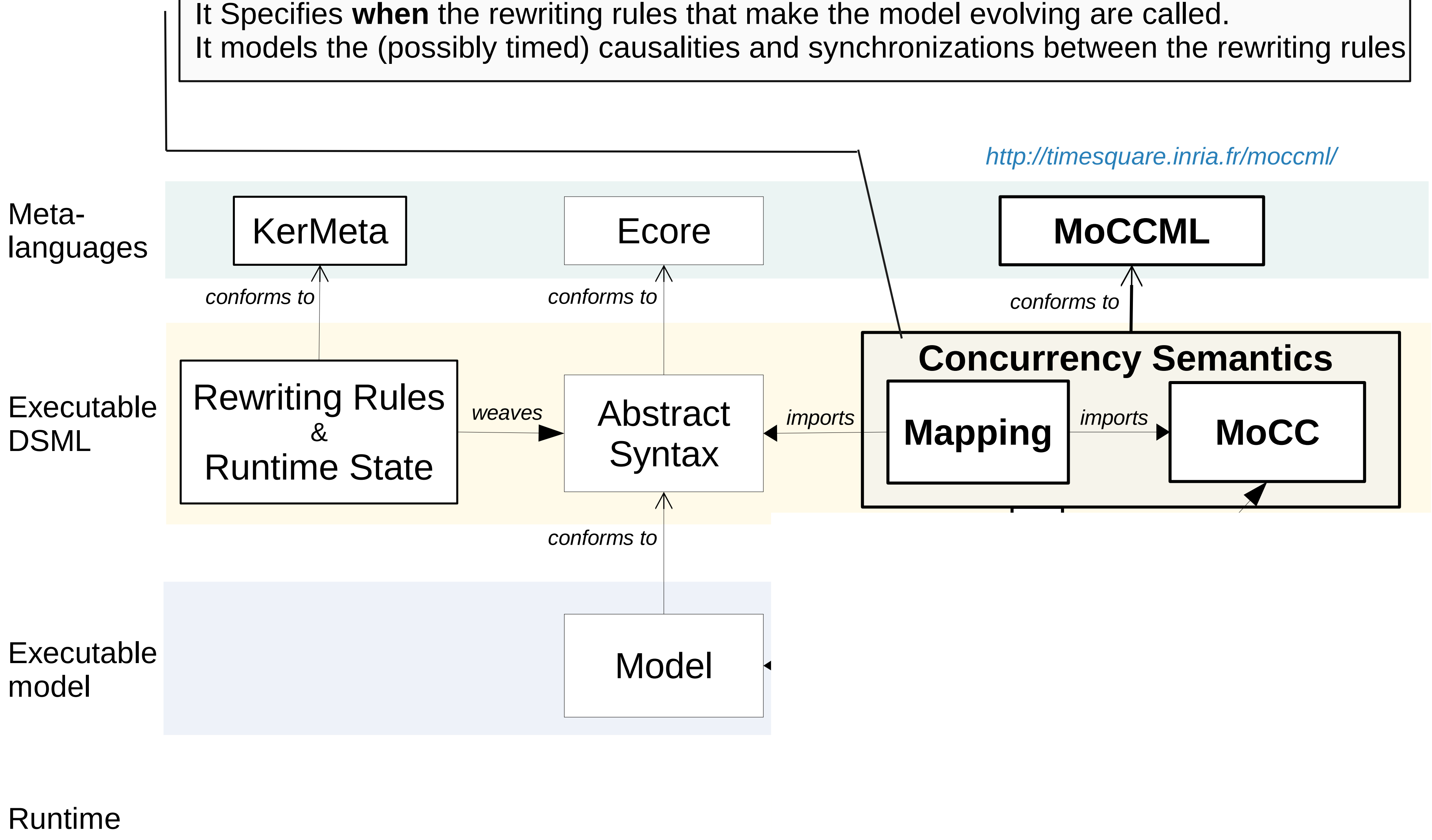
- K3 is an action language built on top of the Xtend programming language and mainly used to implement the execution semantics of Ecore metamodels. Concretely, K3 allows to "re-open" the classes generated from an Ecore metamodel using simple annotations in order to weave new features and operations.
- We can use it to weave the **state** and the **rewriting rules**, *e.g.,*

```
@Aspect(className=While)
class While_EvaluableAspect extends Control_EvaluableAspect {
    def Boolean evaluate() {
        var Boolean resCond = _self.condition.evaluate as Boolean
        return resCond
    }
}

@Aspect(className=While)
class While_ExecutableAspect extends Control_ExecutableAspect {
    def void execute() {
        while (_self.evaluate) {
            _self.block.execute
        }
    }
}
```

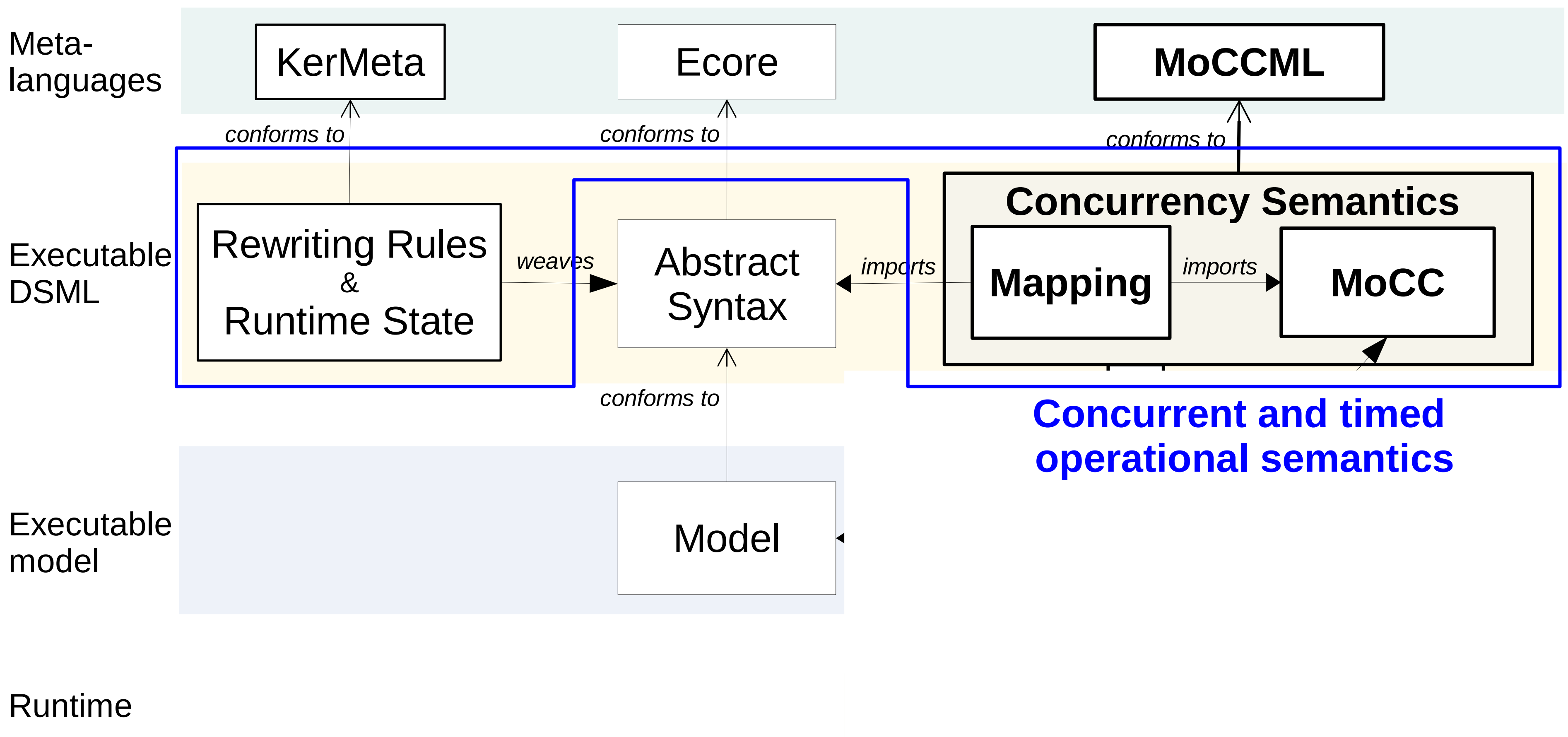
```
@Aspect(className=BooleanVariable)
class BooleanVariableState  {
    public Boolean value

    def Object evaluate(){
        return _self.value
    }
}
```

The GEMOC approach

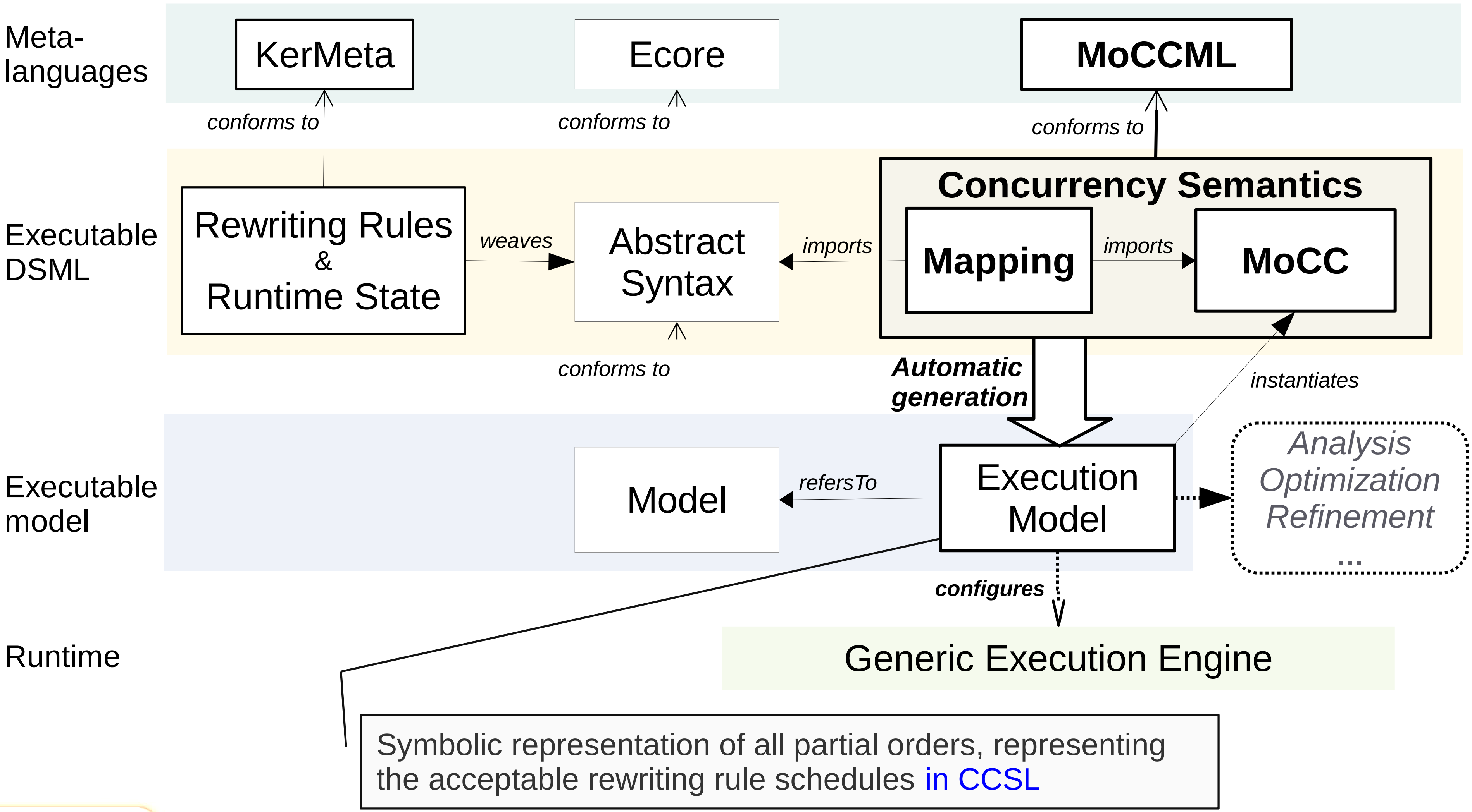
It Specifies **when** the rewriting rules that make the model evolving are called.
It models the (possibly timed) causalities and synchronizations between the rewriting rules



The GEMOC approach

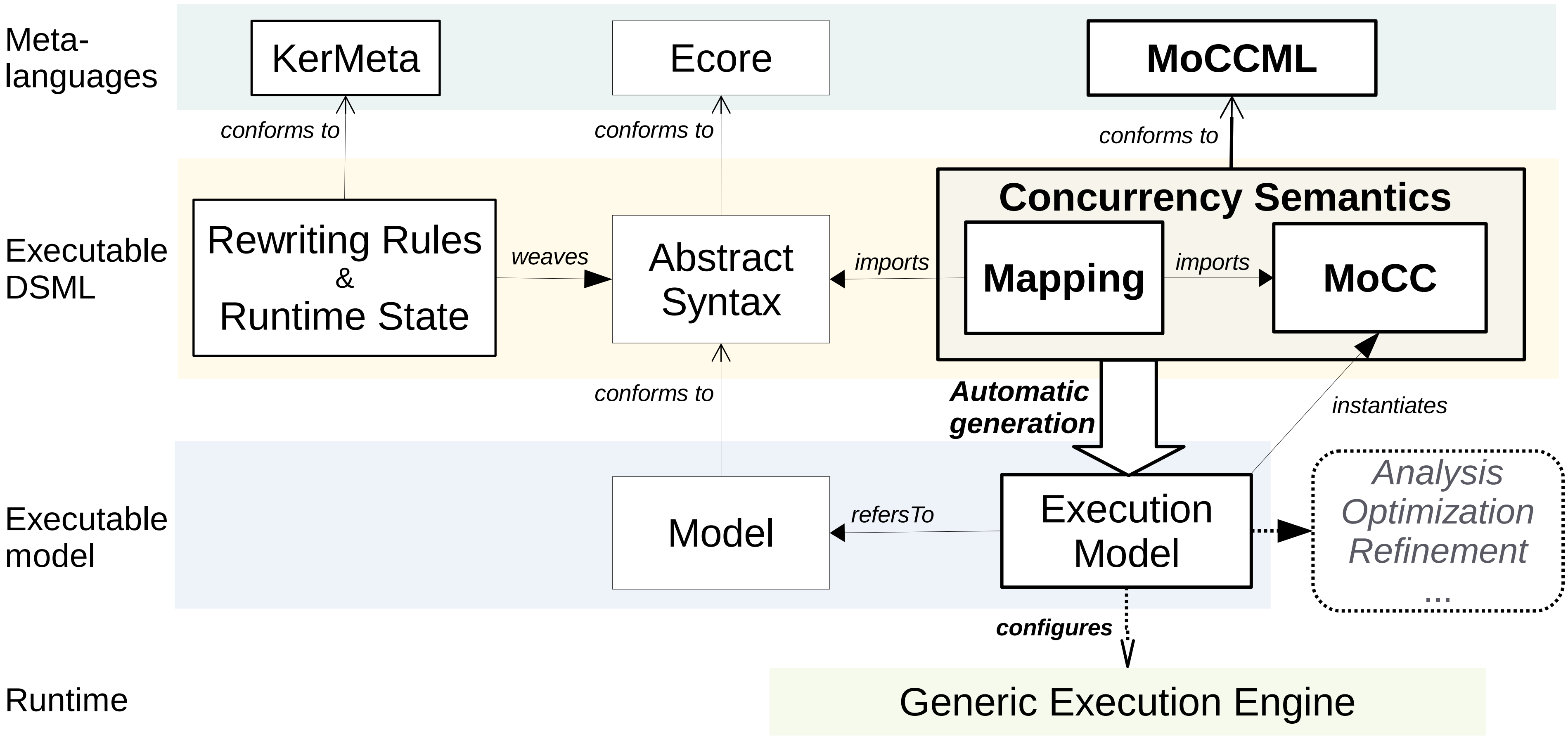


The GEMOC approach

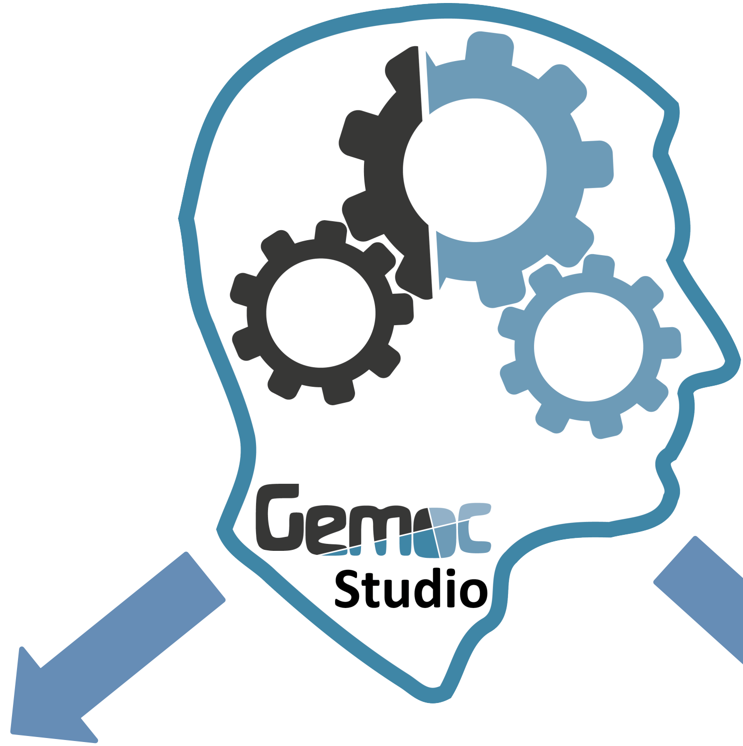


The GEMOC approach

+ graphical concrete syntax in Sirius or Xtext, which uses a meta-language as well



The GEMOC Studio



Research Consortium
http://eclipse.org/gemoc

http://gemoc.org/studio

Language Workbench

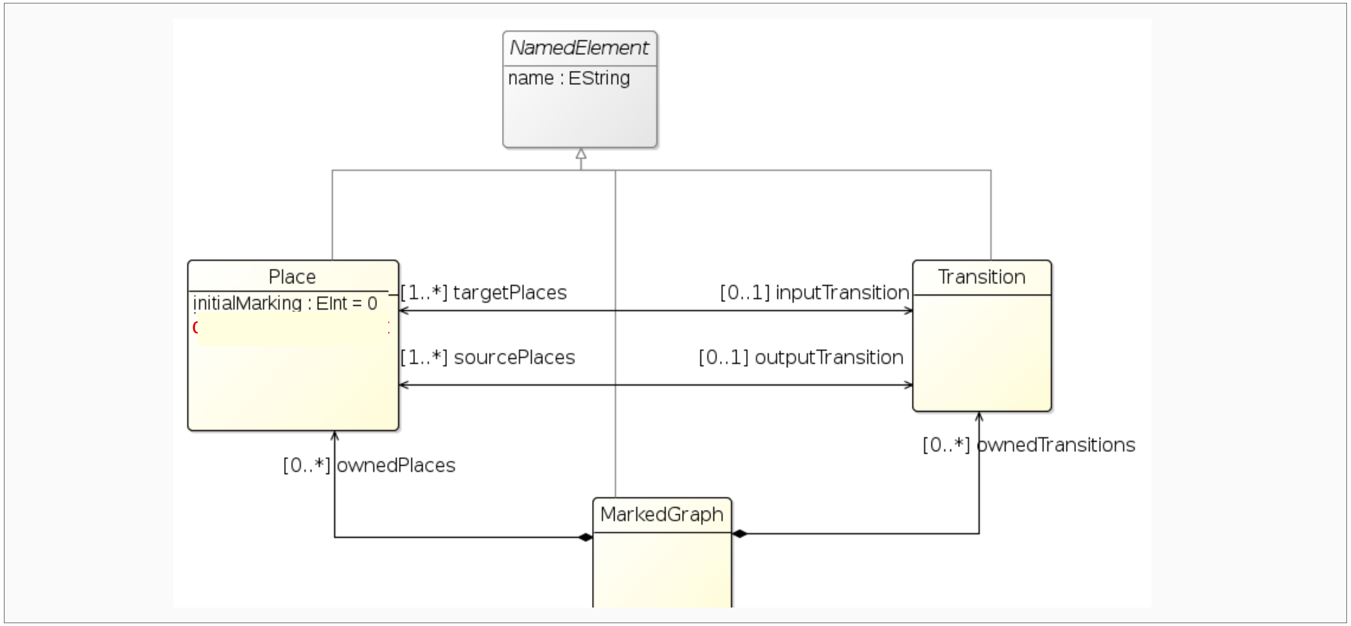
Design and compose your executable DSMLs

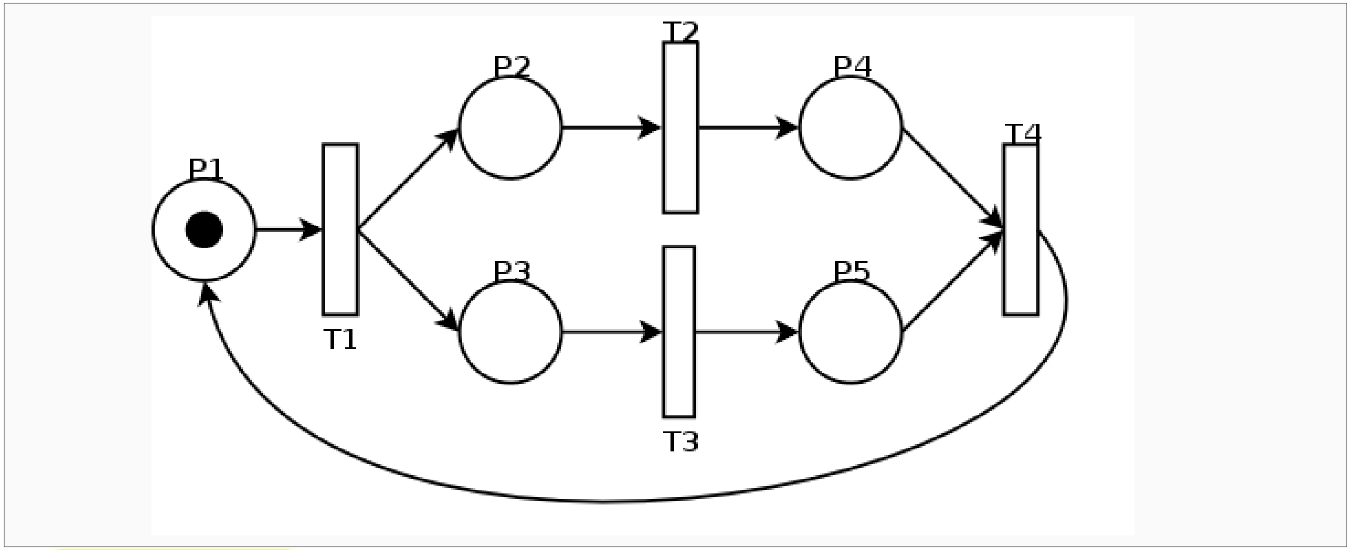
Modeling Workbench

Edit and debug your heterogeneous models

Running example: AS

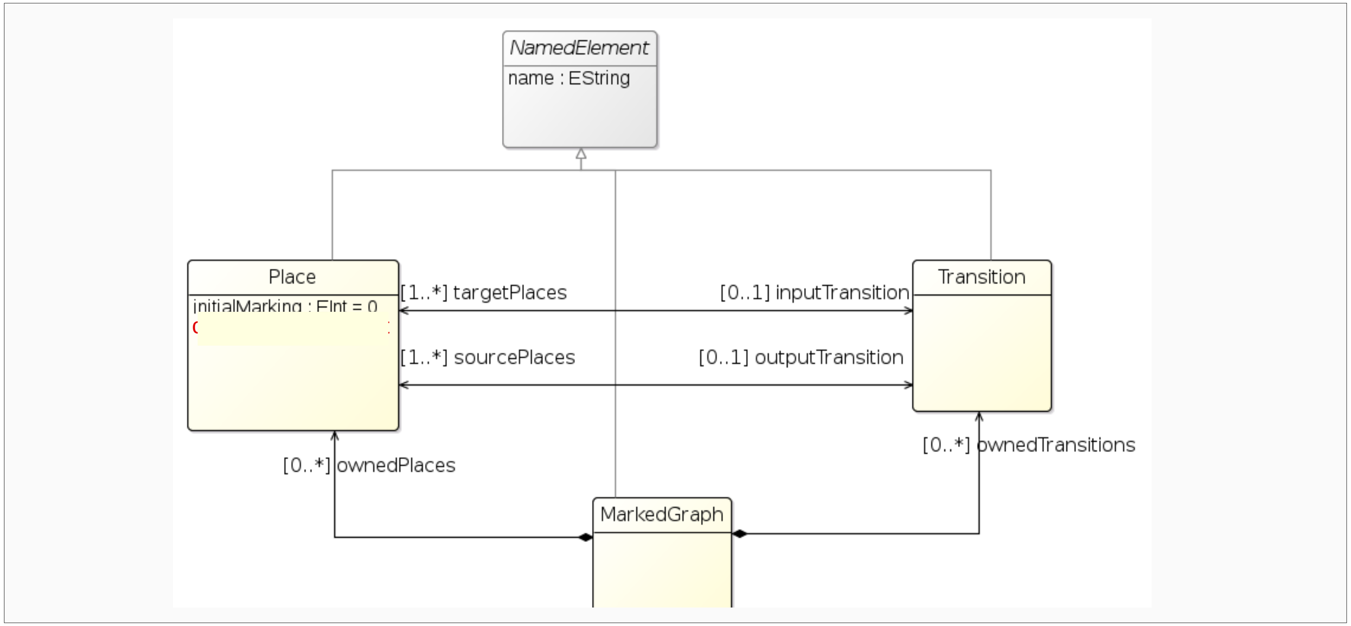
Ecore+Sirius

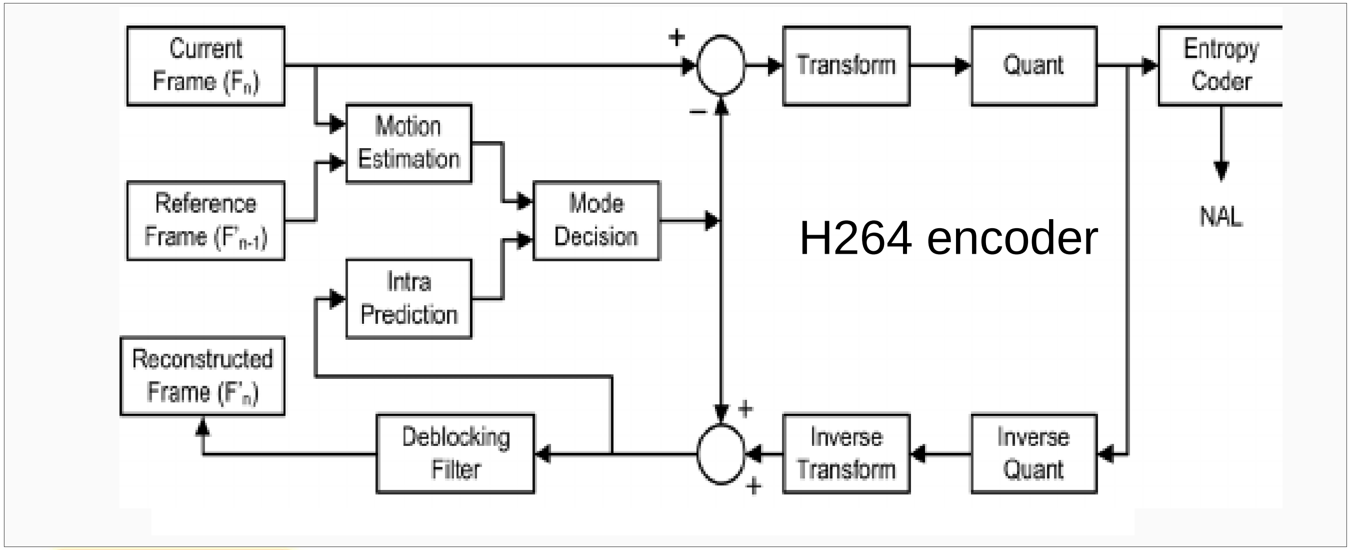




Running example: AS

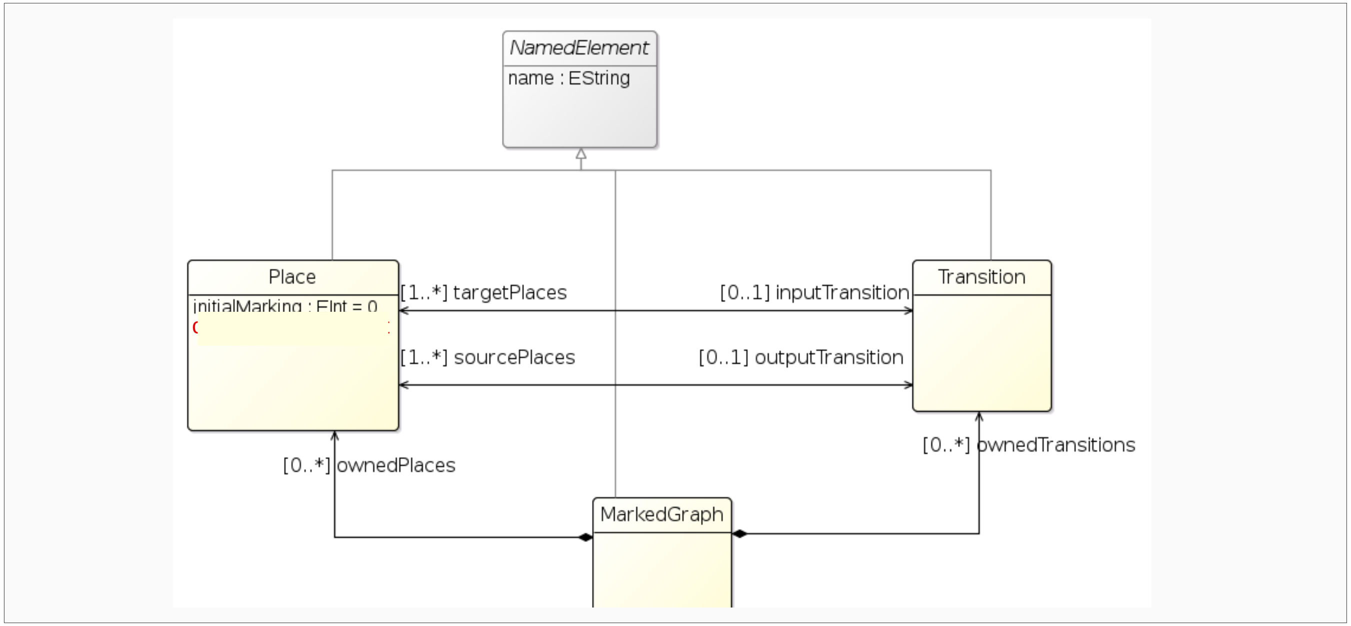
Ecore+Sirius

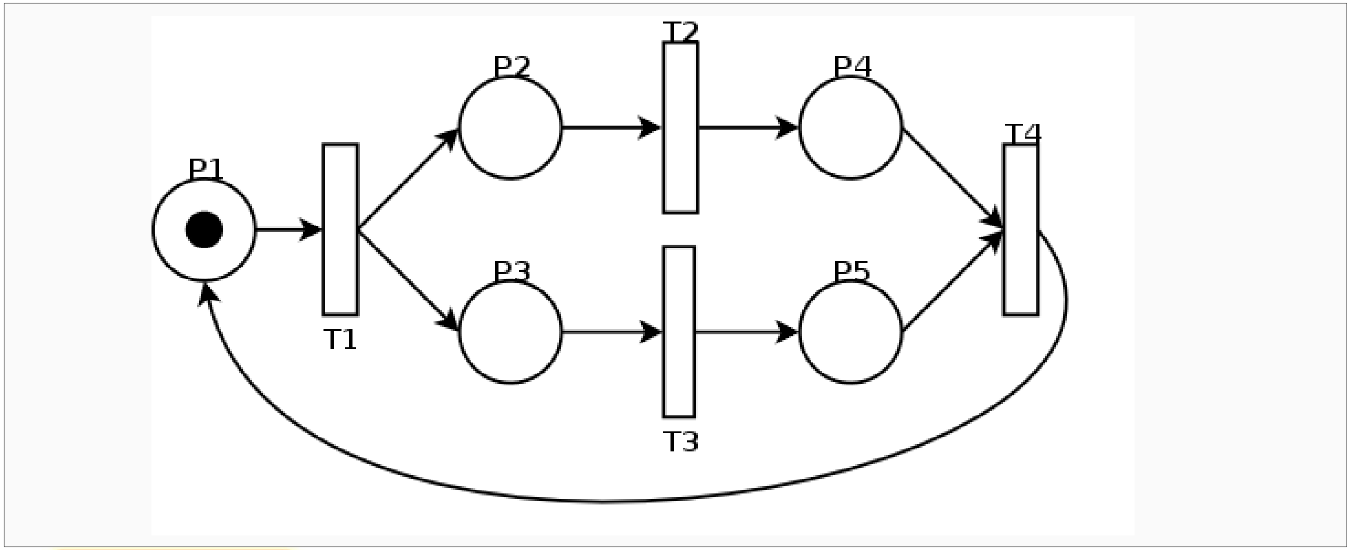




Running example: AS

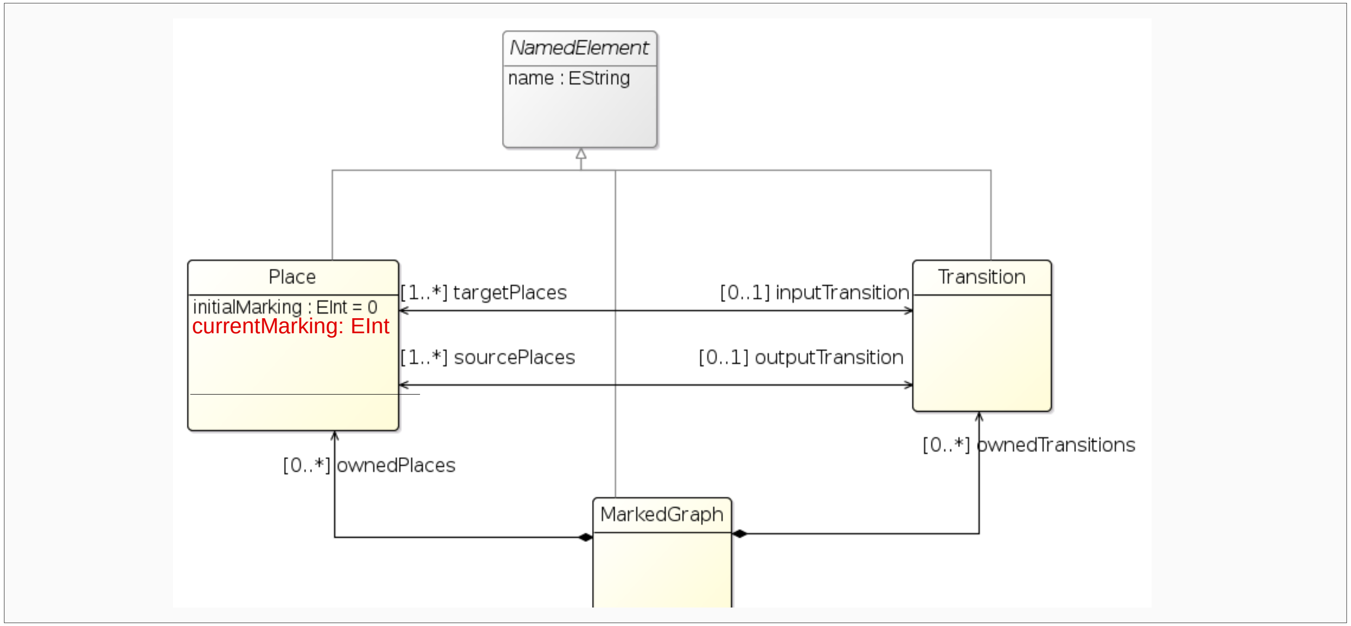
Ecore+Sirius

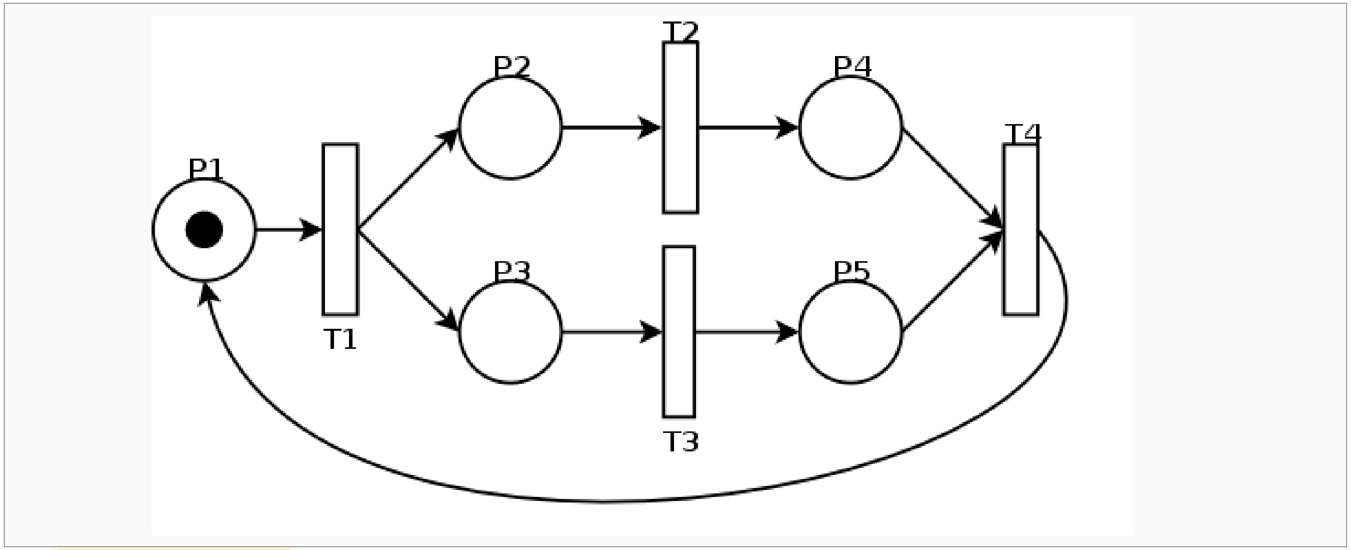




Running example: AS+DSA

Kermeta3





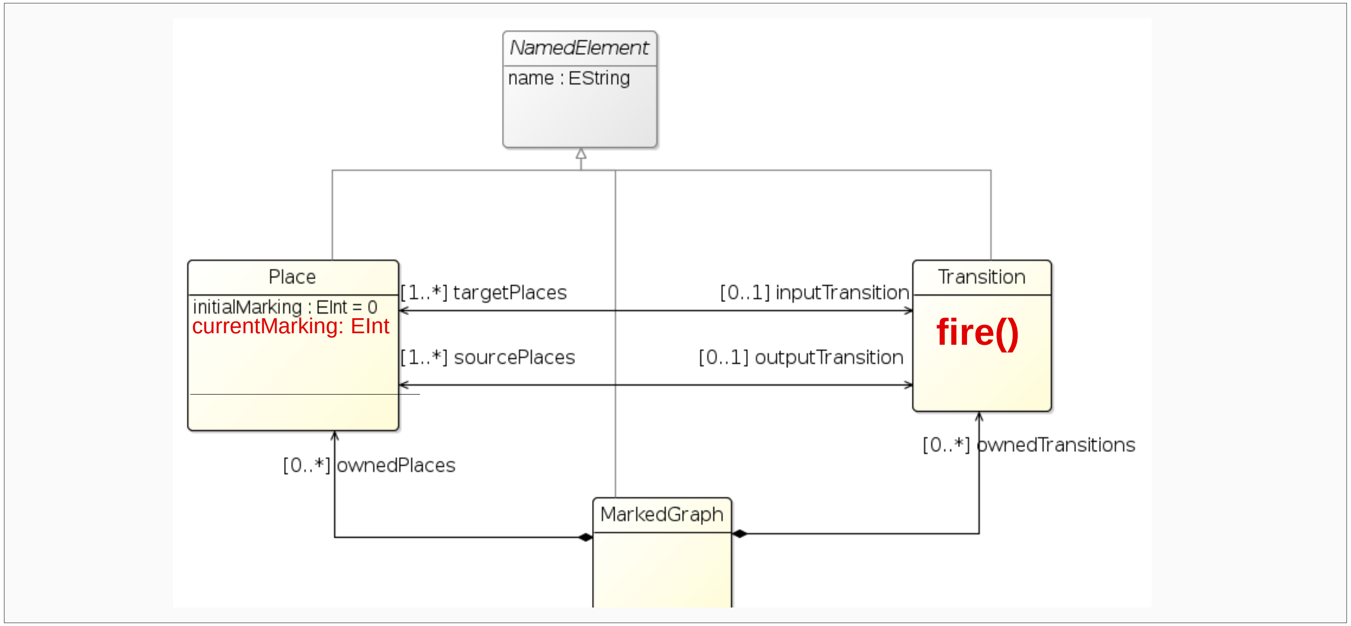
Domain Specific Action

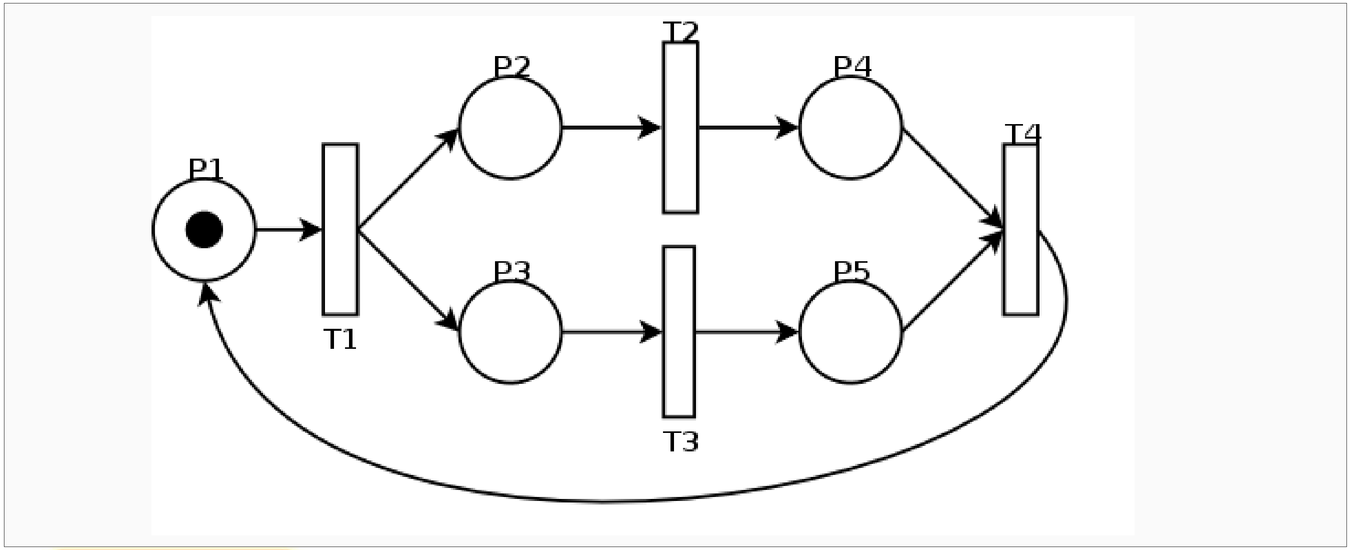
(model state)

- The ***current marking*** represents the runtime state of this simple language

Running example: AS+DSA

Kermeta3





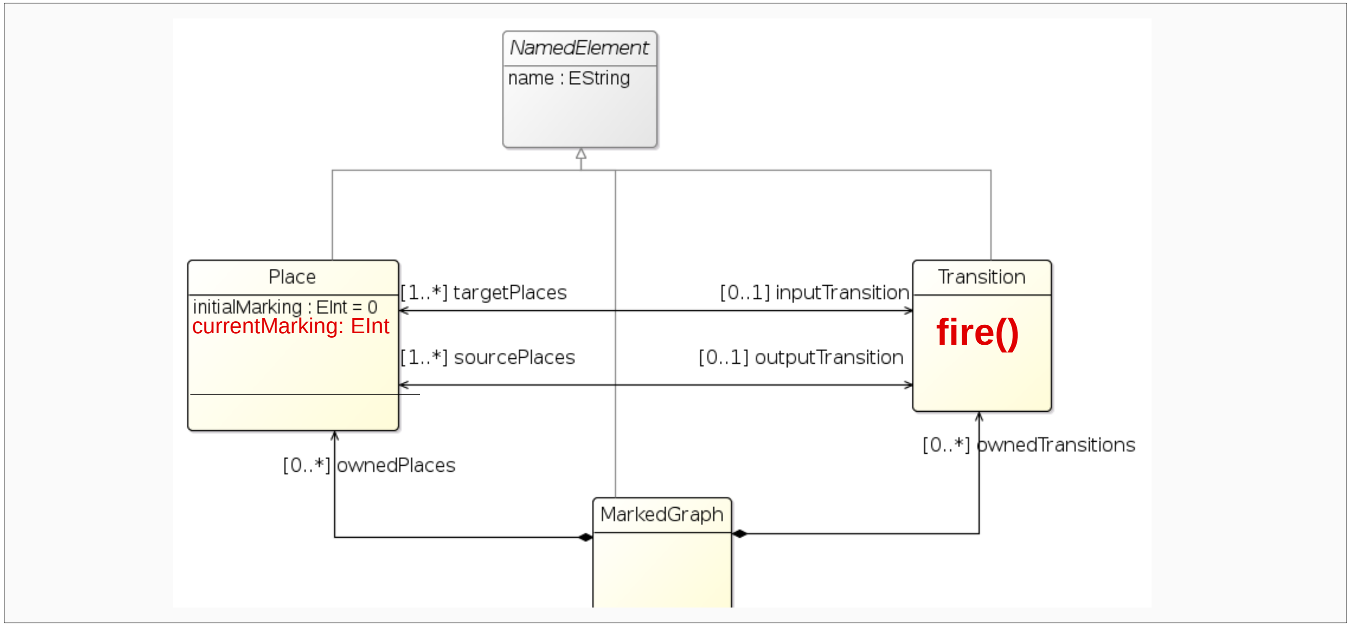
Domain Specific Action

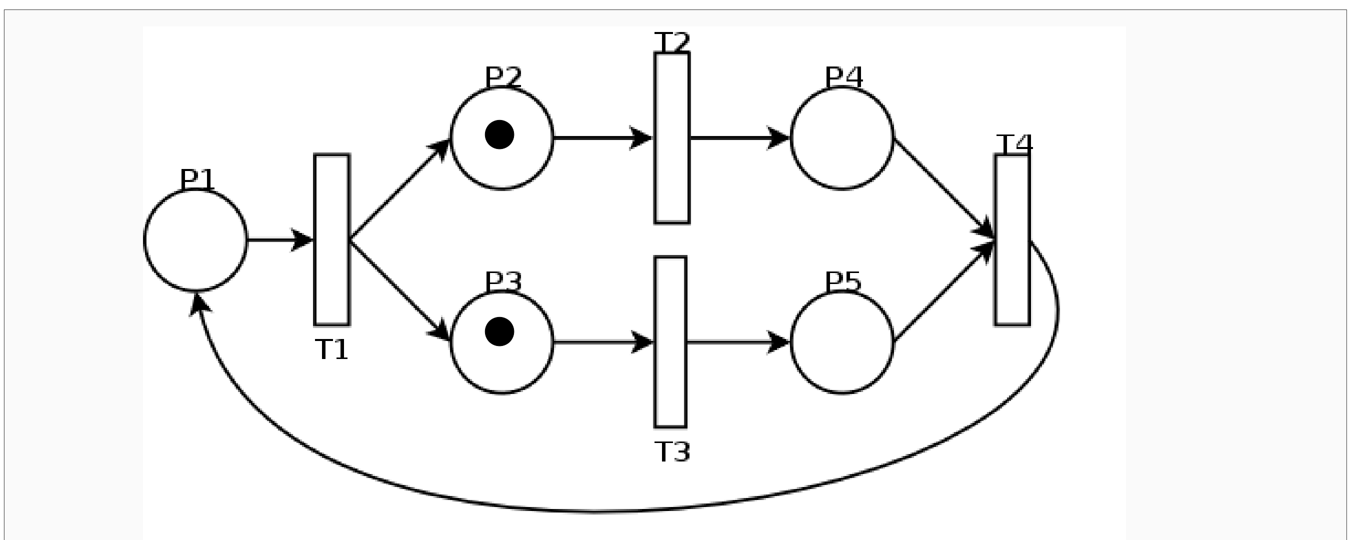
(rewriting rules)

```
def fire(){
    _self.sourcePlaces.forEach [
            currentMarking – –
        ]
    _self.targetPlaces.forEach [
            currentMarking ++
        ]
}
```

Running example: AS+DSA

Kermeta3





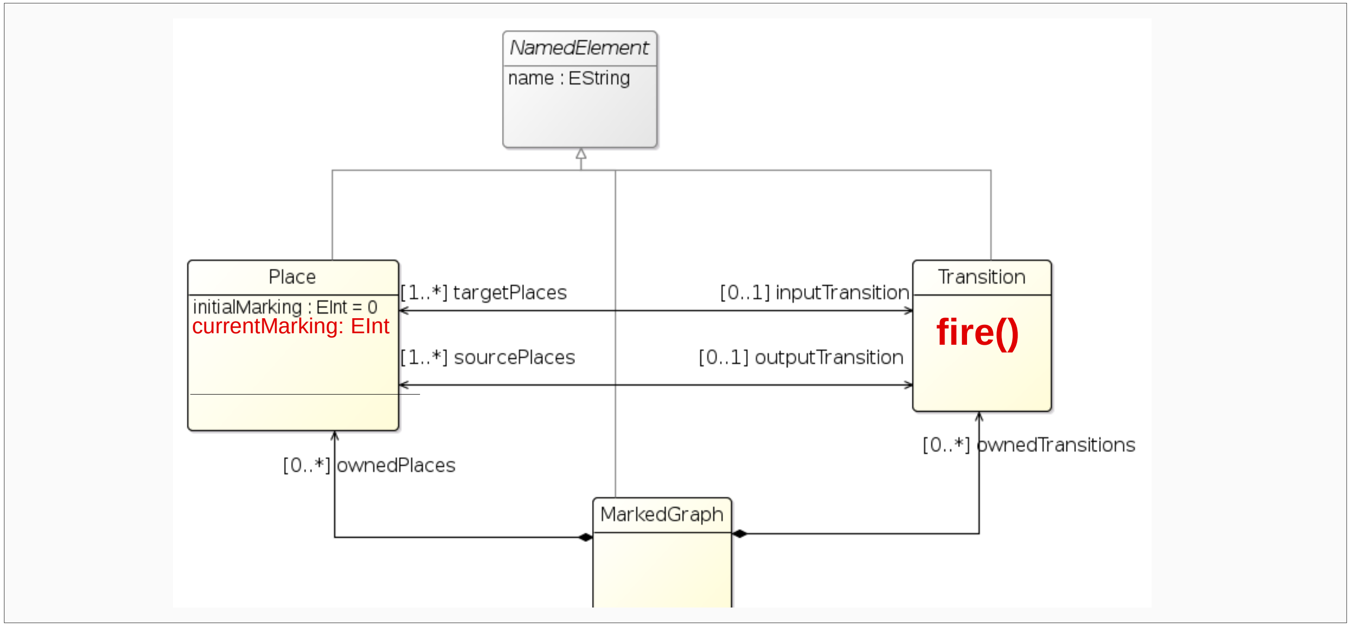
Domain Specific Action

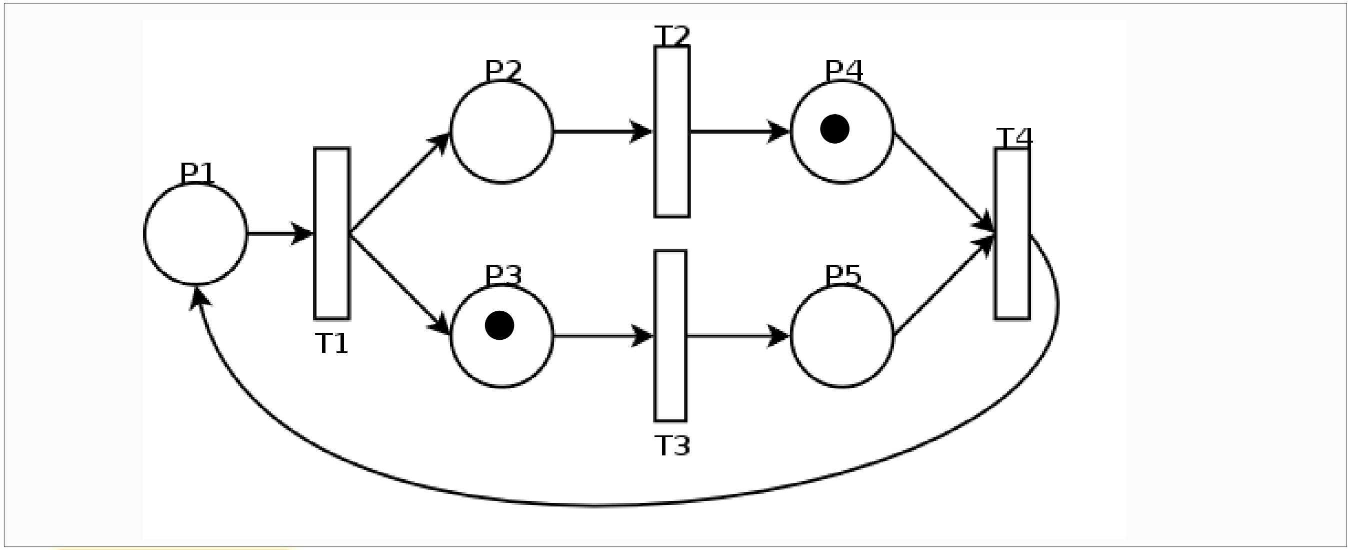
(rewriting rules)

```
def fire(){
    _self.sourcePlaces.forEach [
            currentMarking – –
        ]
    _self.targetPlaces.forEach [
            currentMarking ++
        ]
}
```

Running example: AS+DSA

Kermeta3





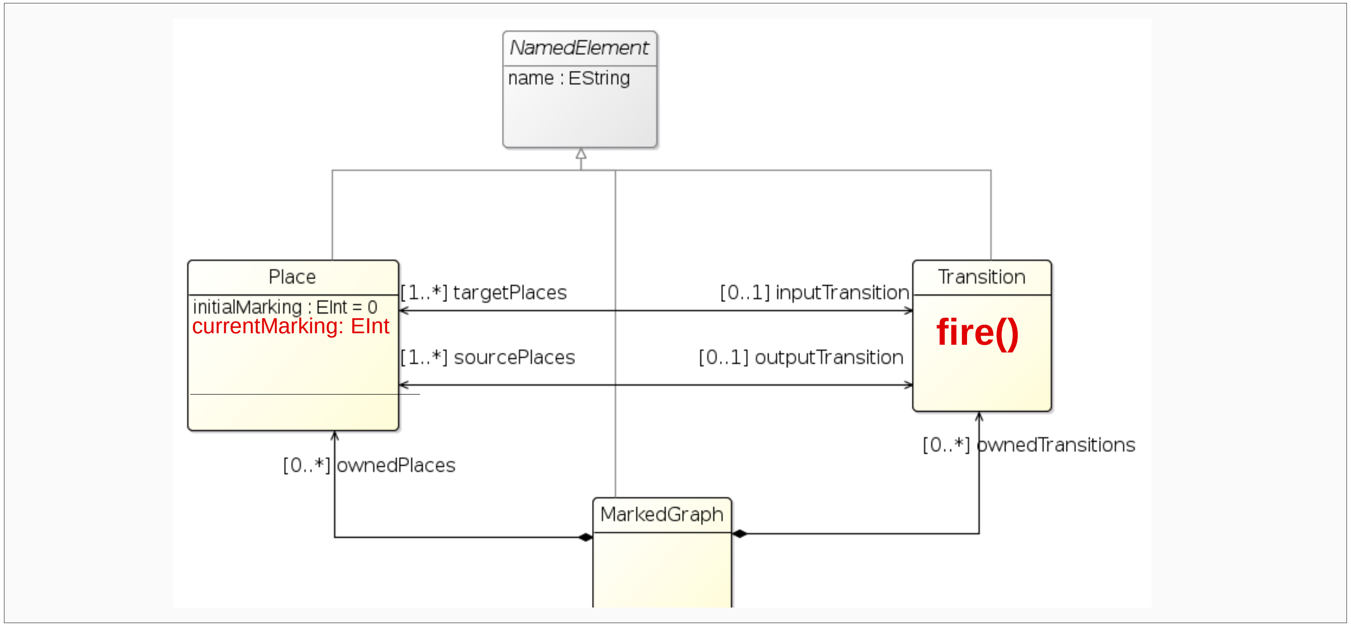
Domain Specific Action

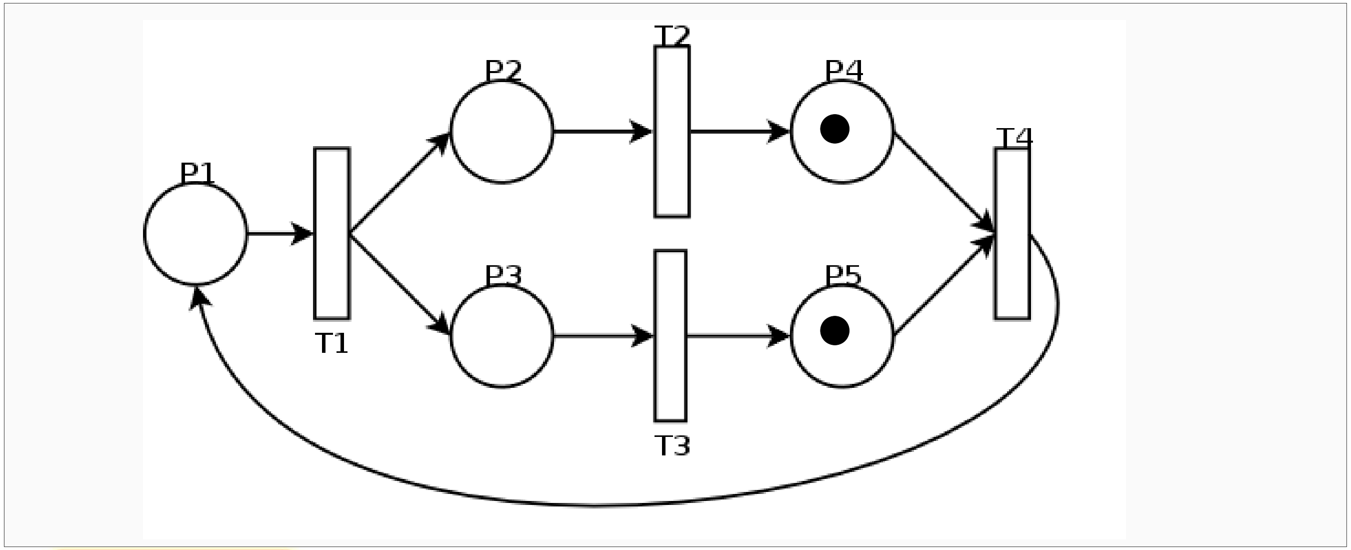
(rewriting rules)

```
def fire(){
    _self.sourcePlaces.forEach [
                currentMarking – –
            ]
    _self.targetPlaces.forEach [
                currentMarking ++
            ]
}
```

Running example: AS+DSA

Kermeta3





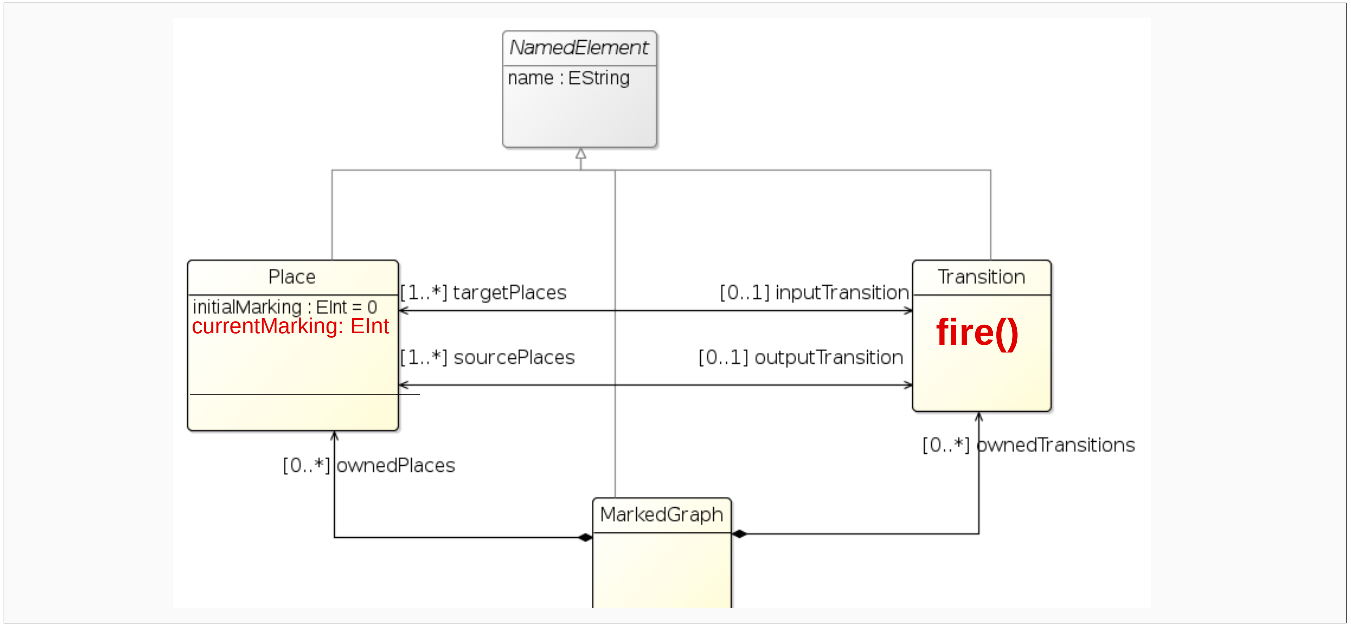
Domain Specific Action

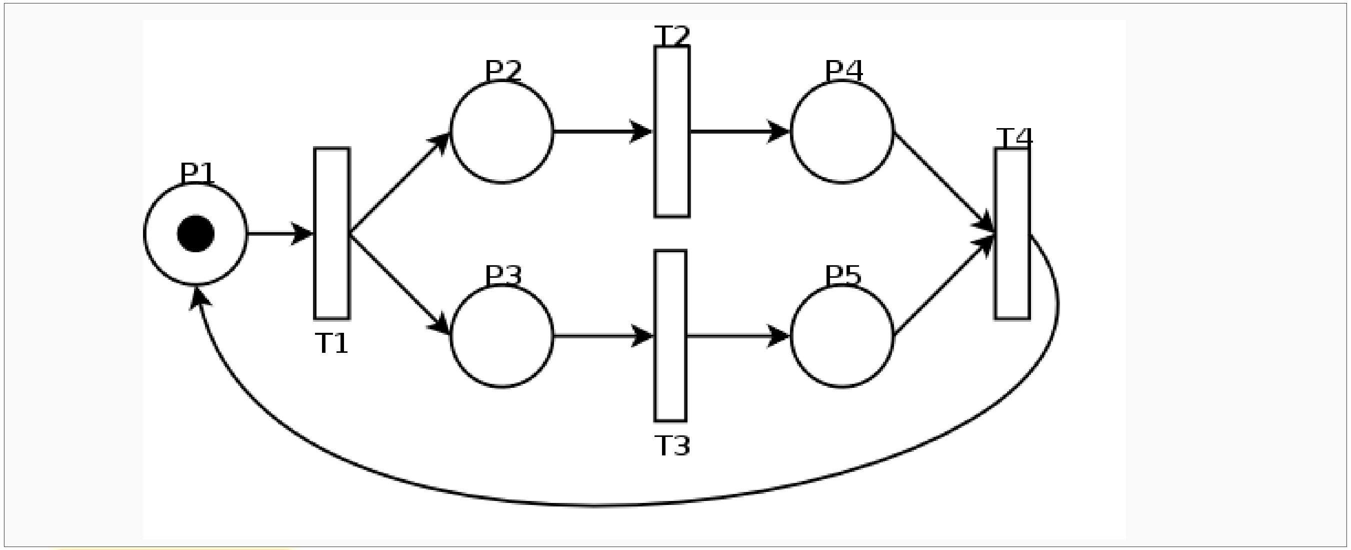
(rewriting rules)

```
def fire(){
    _self.sourcePlaces.forEach [
                currentMarking – –
            ]
    _self.targetPlaces.forEach [
                currentMarking ++
            ]
}
```

Running example:
AS+DSA

Kermeta3





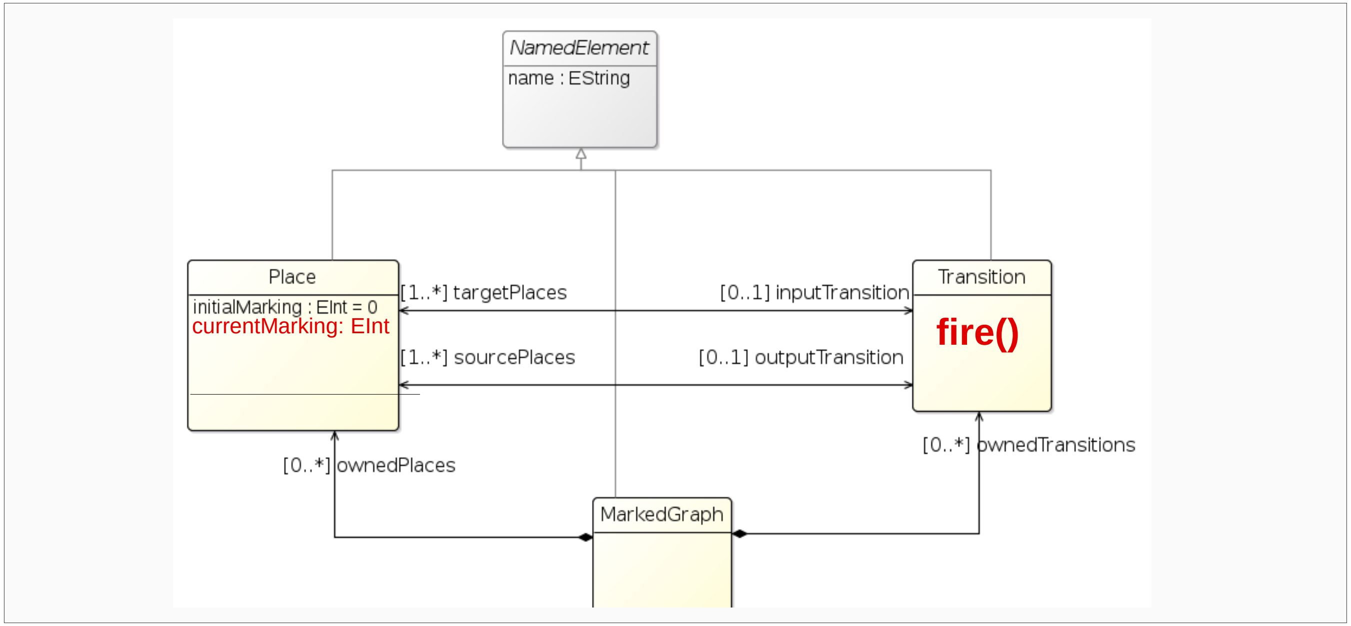
Domain Specific Action

(rewriting rules)

```
def fire(){
    _self.sourcePlaces.forEach [
            currentMarking – –
        ]
    _self.targetPlaces.forEach [
            currentMarking ++
        ]
}
```

Running example: AS+DSA

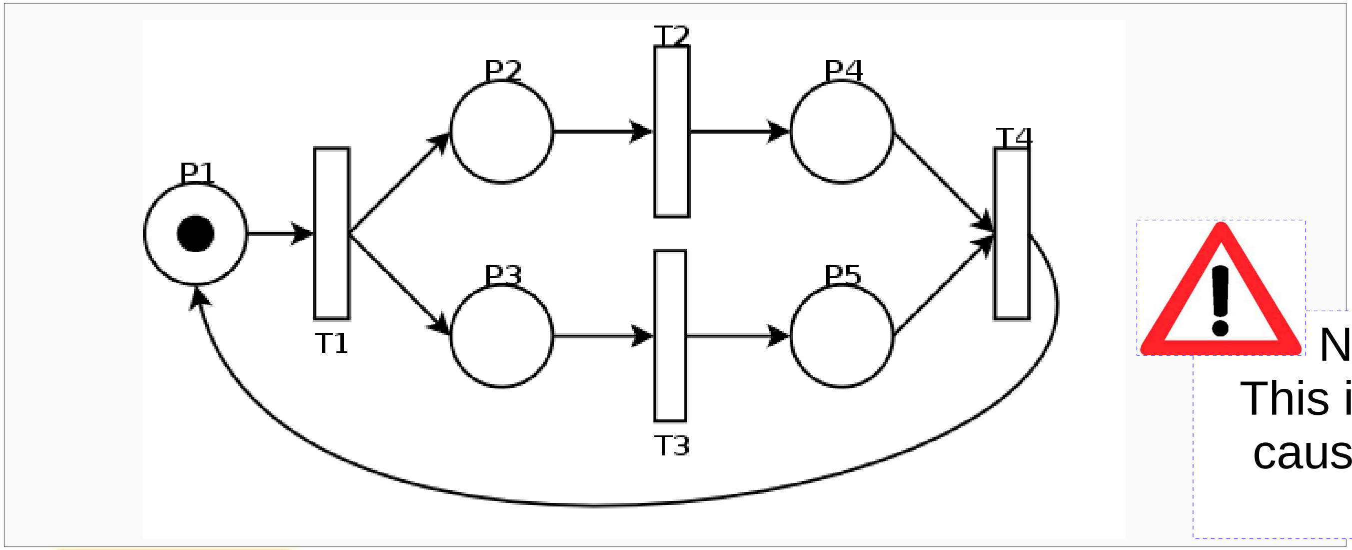
Kermeta3



Domain Specific Action

(rewriting rules)

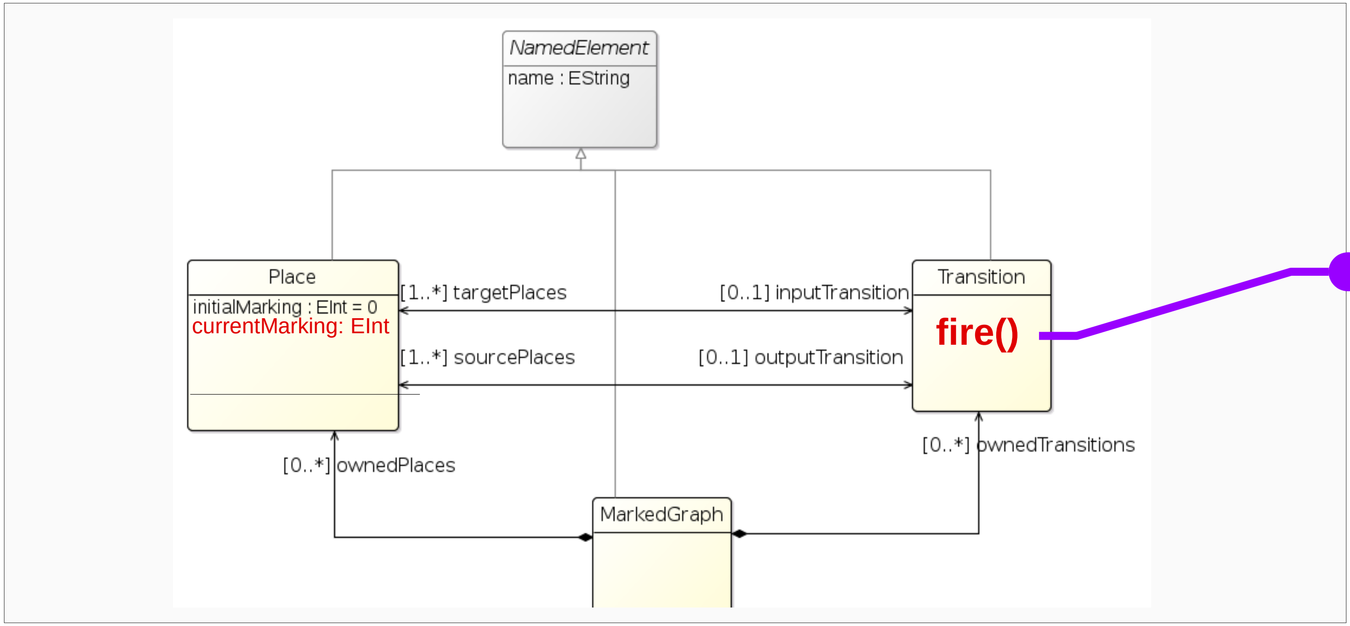
```
def fire(){
    _self.sourcePlaces.forEach [
            currentMarking – –
        ]
    _self.targetPlaces.forEach [
            currentMarking ++
        ]
}
```



⚠ Nobody calls the *fire()* operation. This is the model of concurrency and causality that specifies **when** things happen

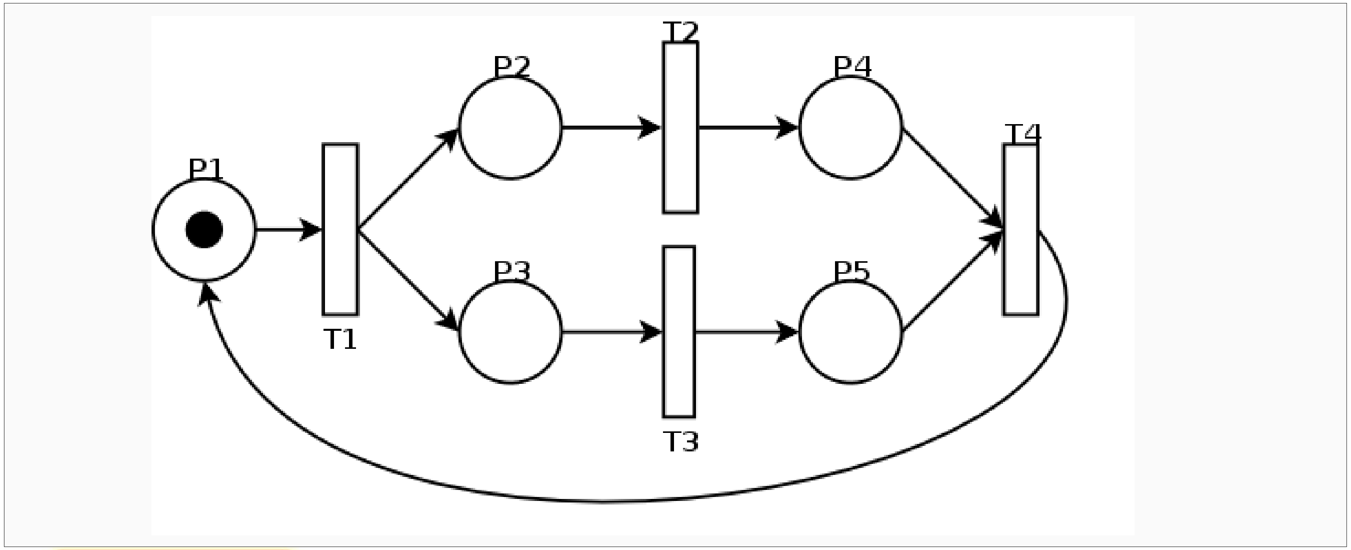
Running example: AS+DSA+DSE

ECL



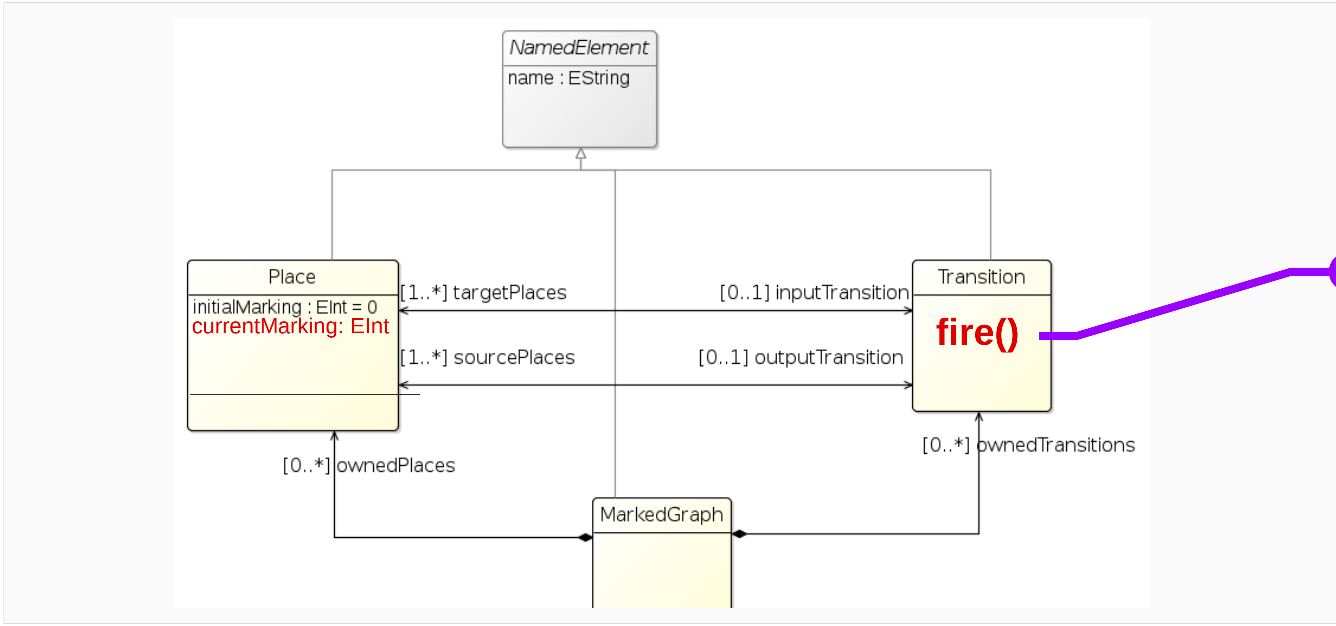
fireIt: DSE

Domain Specific Events act as “handles” to the DSA



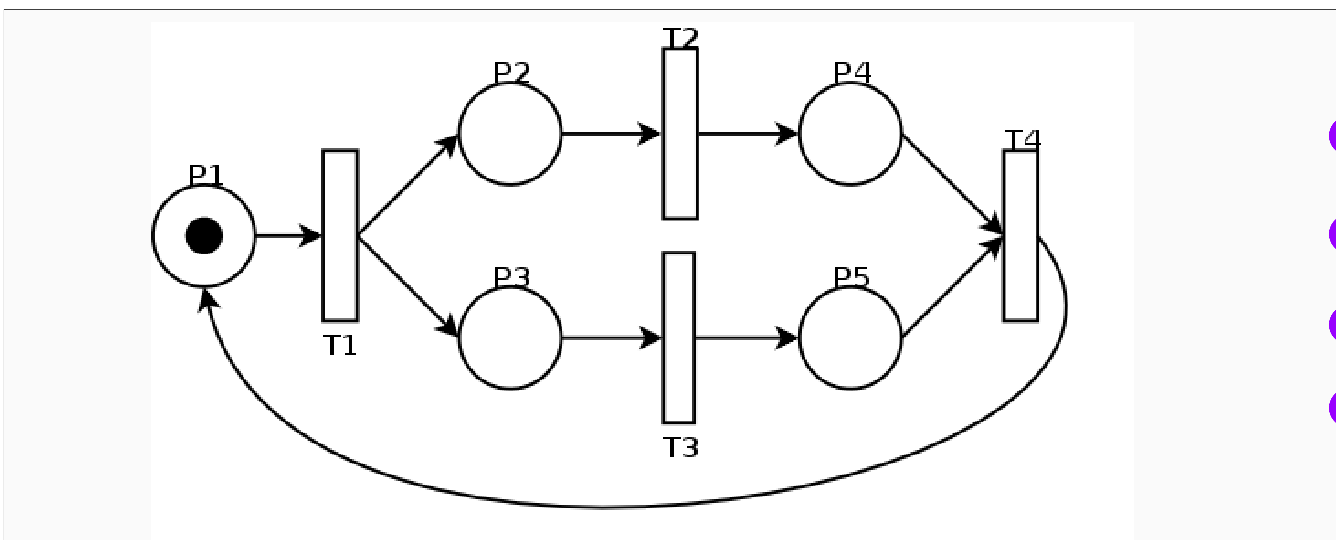
Running example: AS+DSA+DSE

MoCCML mapping (ex ECL)



- fireIt: DSE

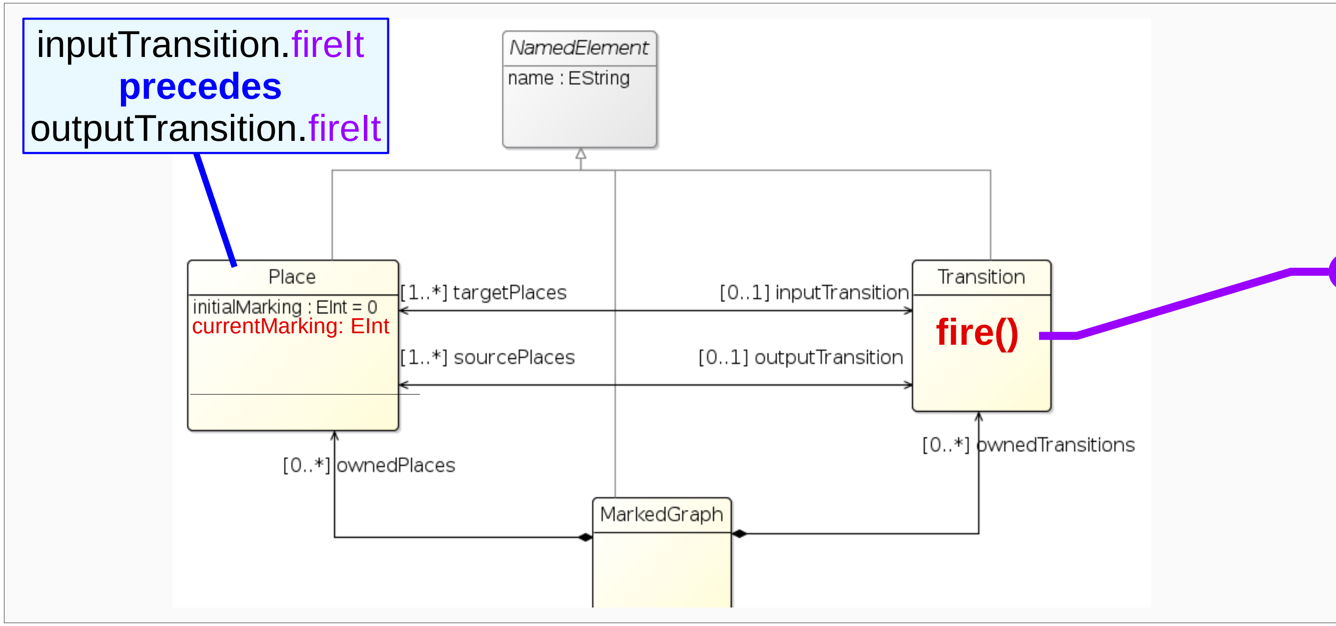
Domain Specific Events act as “handles” to the DSA



- fire_T1: fireIt
- fire_T2: fireIt
- fire_T3: fireIt
- fire_T4: fireIt

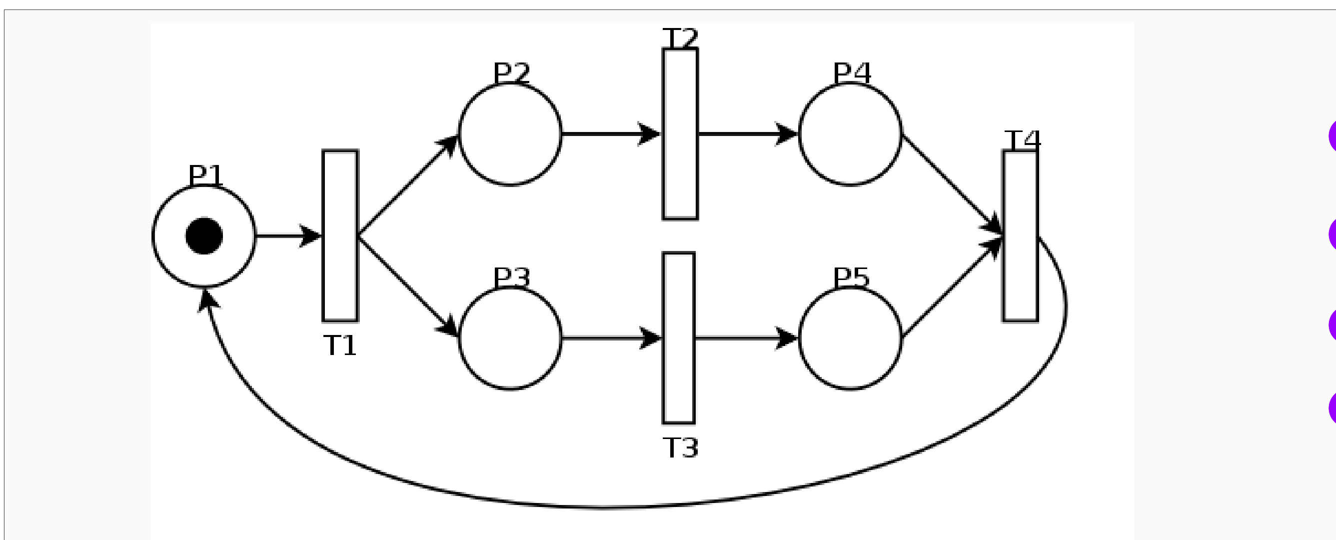
Running example: AS+DSA+DSE+MoCC

MoCCML



- fireIt: DSE

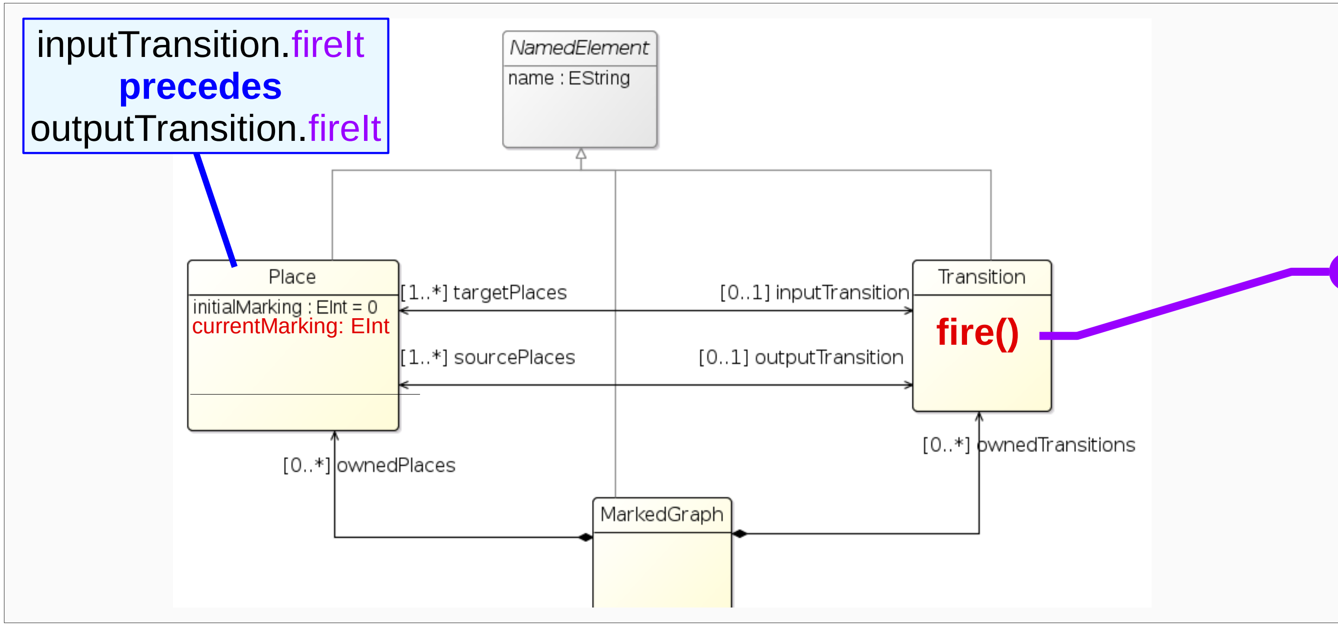
The MoCC constrains the DSE and consequently defines the acceptable schedules of the actions



- fire_T1: fireIt
- fire_T2: fireIt
- fire_T3: fireIt
- fire_T4: fireIt

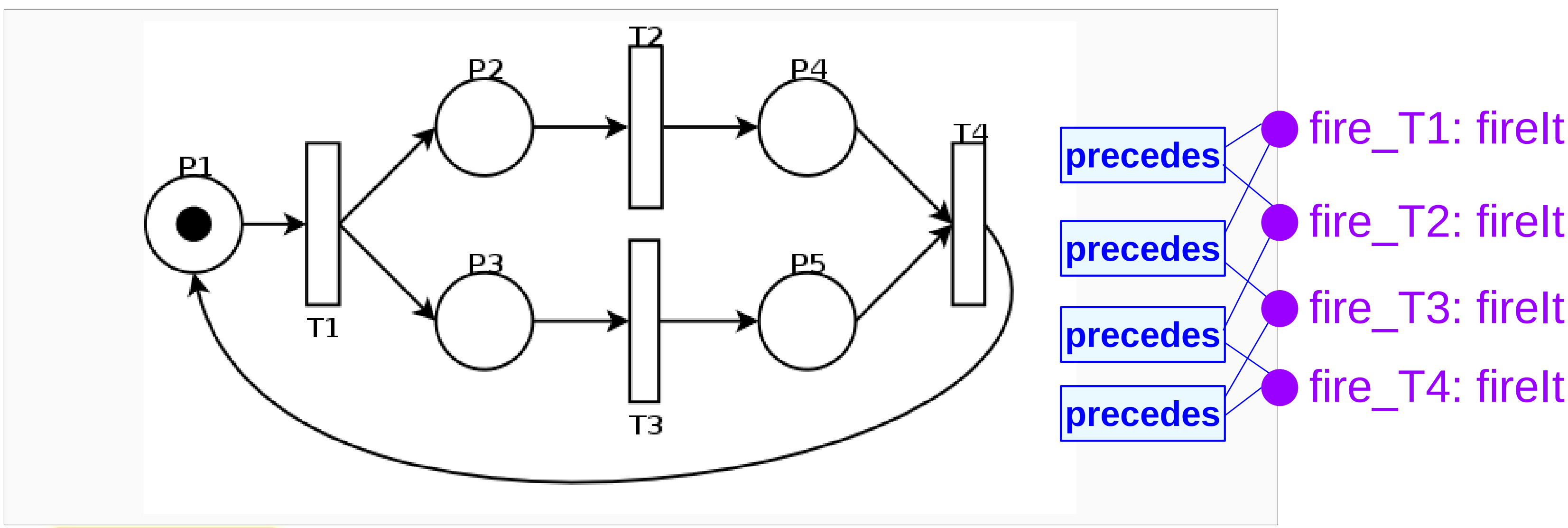
Running example: AS+DSA+DSE+MoCC

MoCCML



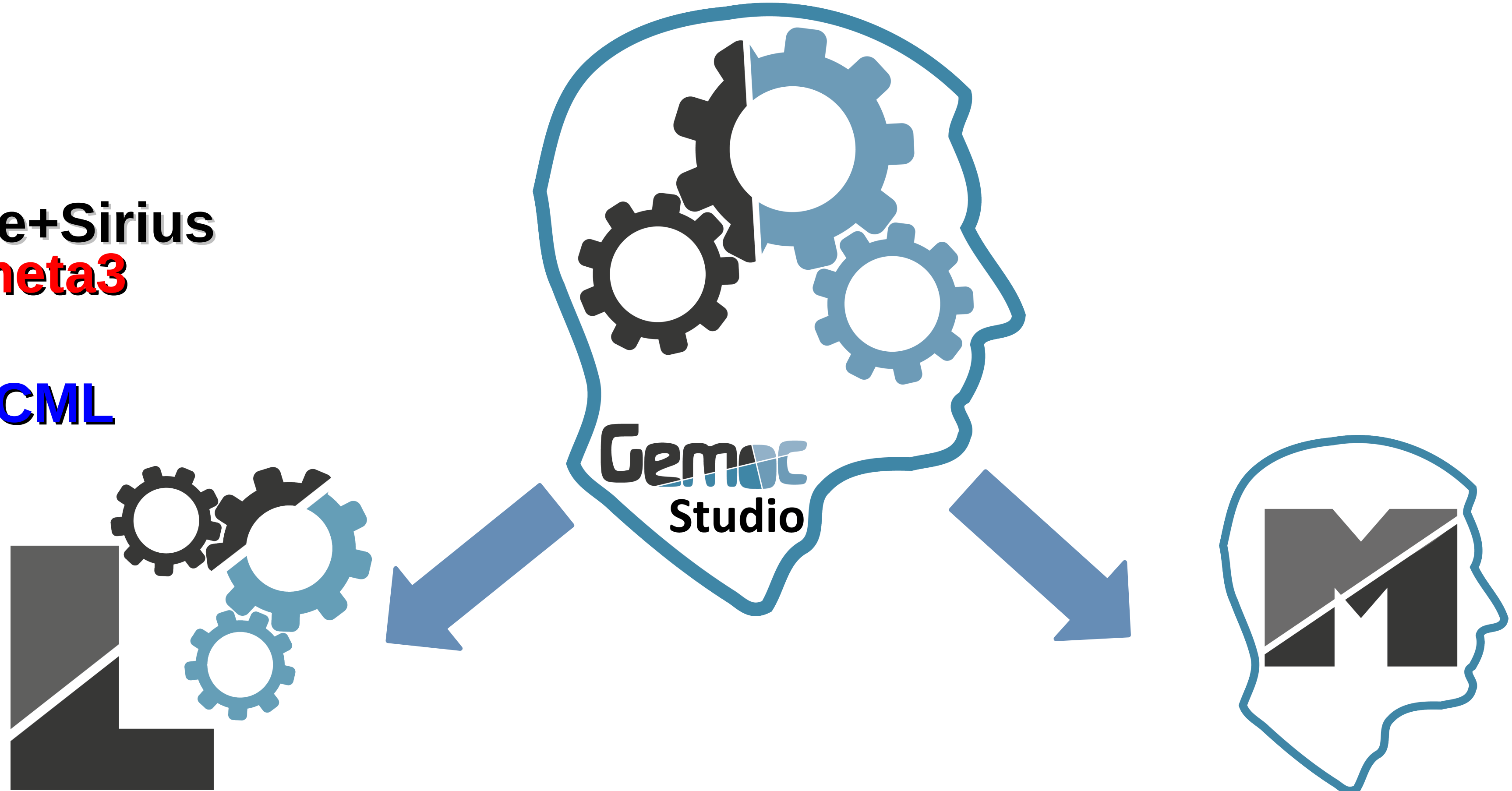
fireIt: DSE

The MoCC constrains the DSE and consequently defines the acceptable schedules of the actions



The GEMOC Studio

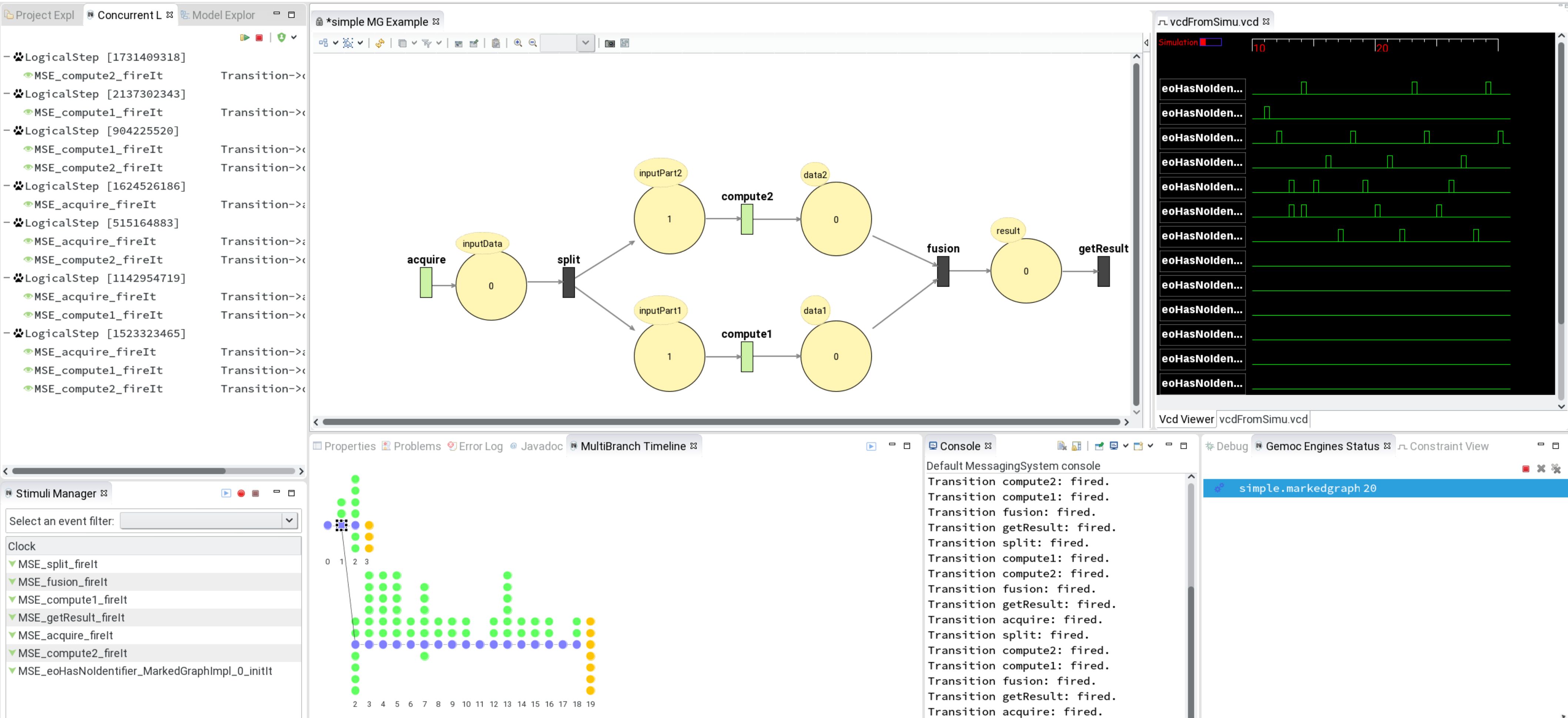
Ecore+Sirius
Kermeta3
ECL
MoCCML



Automatic generation

Language Workbench → **Modeling Workbench**

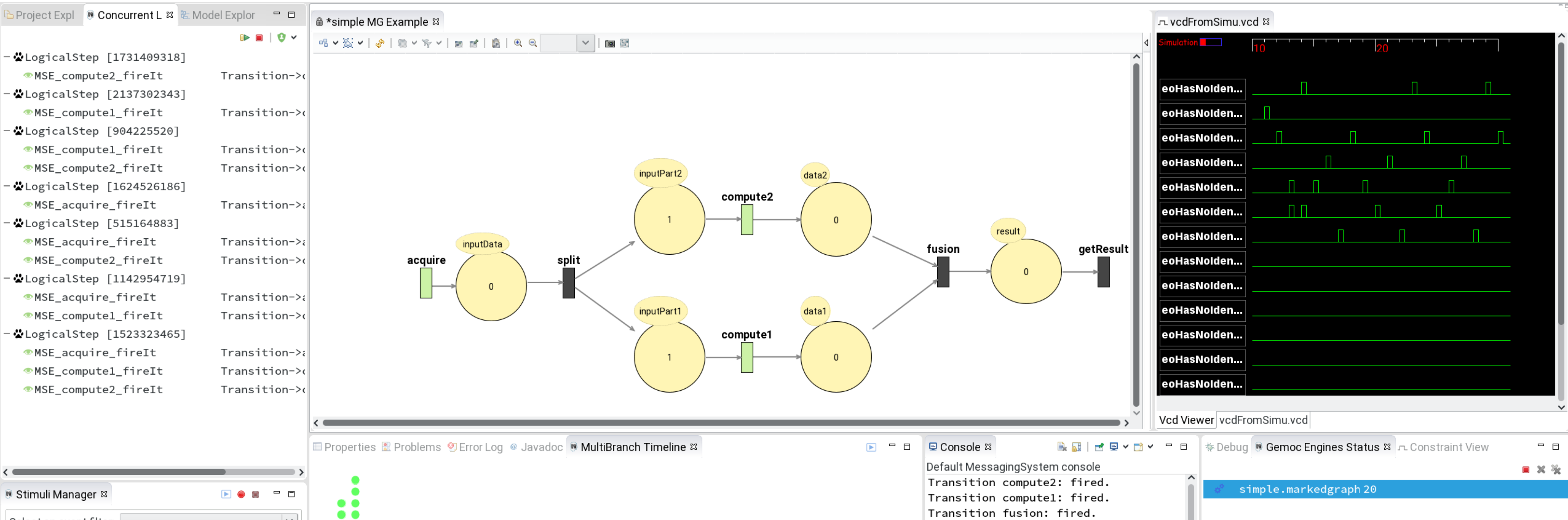
The GEMOC Studio

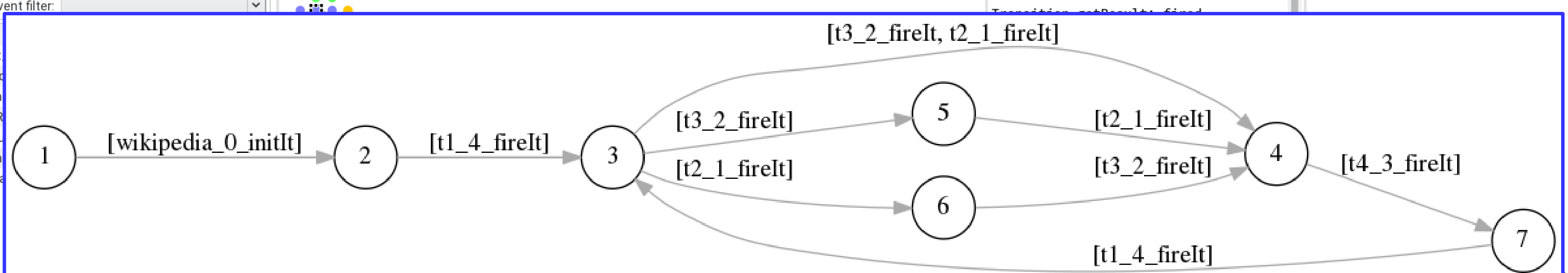


Language Workbench —— Automatic generation ——> **Modeling Workbench**

The GEMOC Studio

SUPPORTED BY ANR





Language Workbench

Modeling Workbench