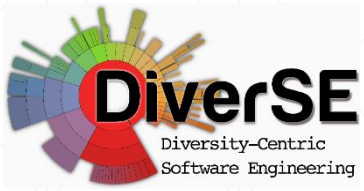


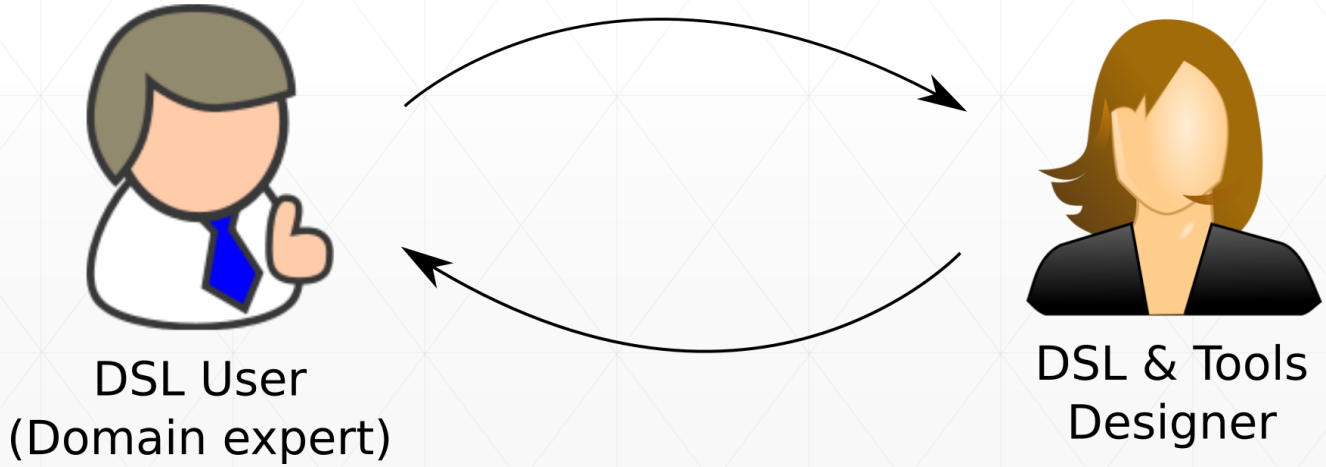
DSL Engineering with Language Interfaces & Algebra

Thomas Degueule, Benoit Combemale, Arnaud Blouin

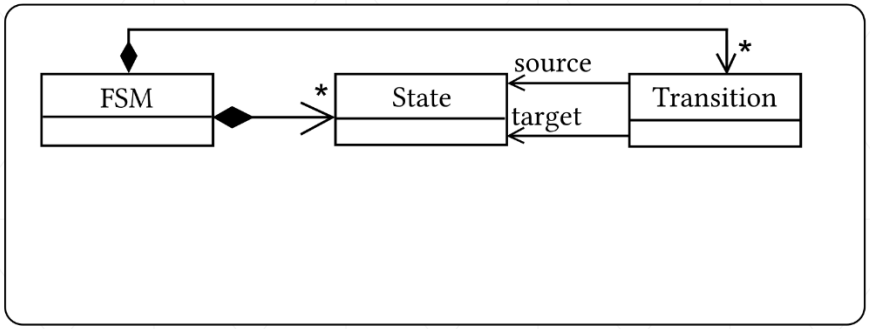
Olivier Barais, Jean-Marc Jezequel

The daily life of DSLs

- Closely evolve with the domain and the experts' understanding of the domain
- Meant for rapid prototyping, evolution
- Extended, shrunk, customized, replaced with alternatives



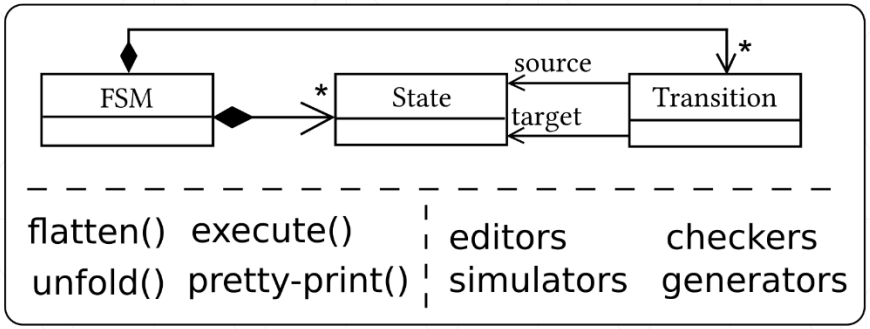
FSM Modeling Environment



produces

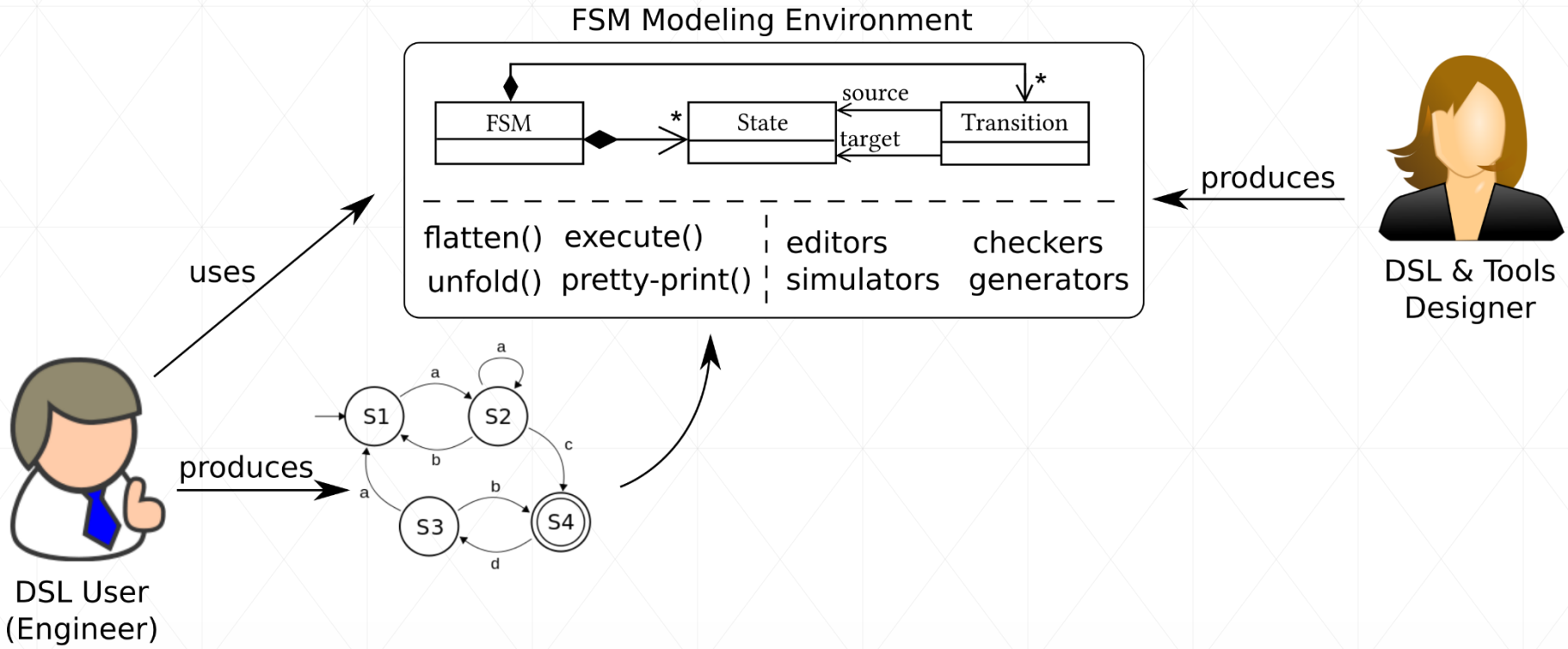
DSL & Tools Designer

FSM Modeling Environment

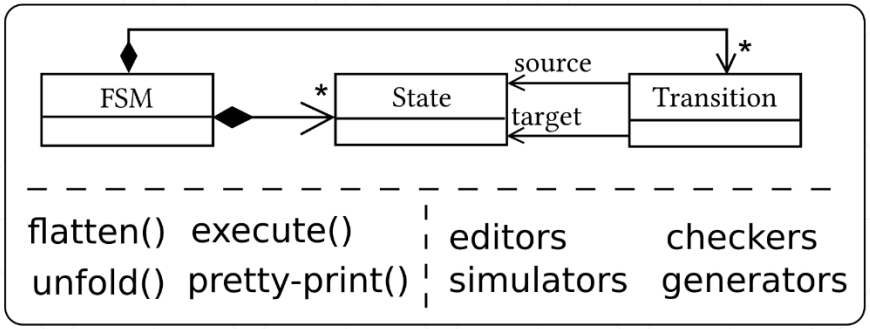


produces

DSL & Tools Designer


FSM Modeling Environment
FSM
State
Transition
source
target
*
*
flatten()
execute()
unfold()
pretty-print()
editors
checkers
simulators
generators
produces
DSL & Tools
Designer
uses
produces
S1
S2
S3
S4
a
b
c
d
DSL User
(Engineer)

FSM Modeling Environment



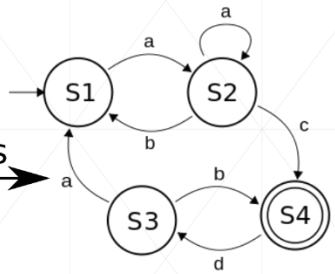
produces

DSL & Tools Designer

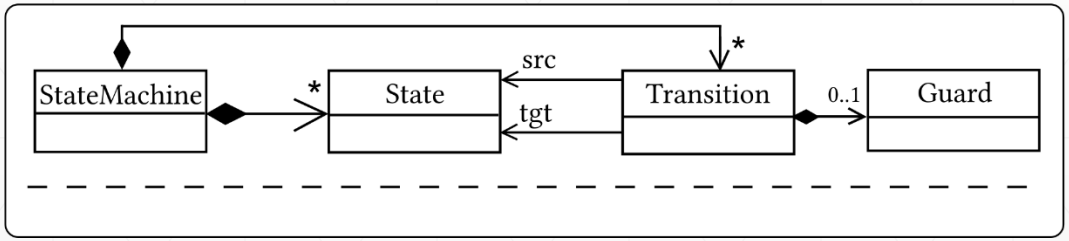
uses

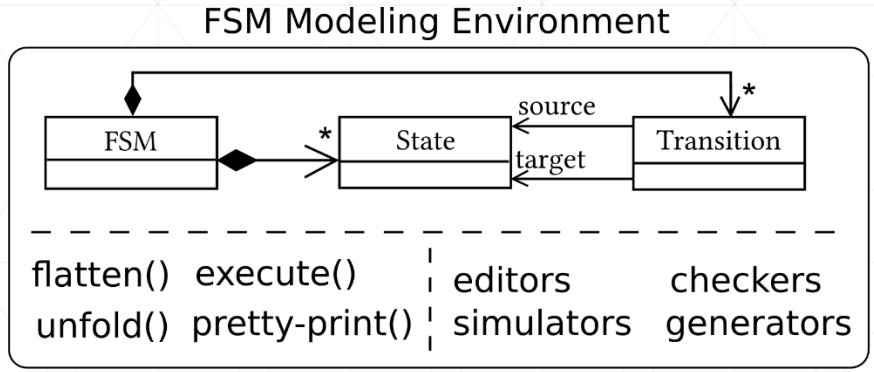
DSL User (Engineer)

produces



Yet Another FSM Modeling Environment





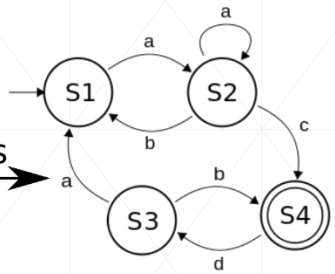
DSL & Tools Designer

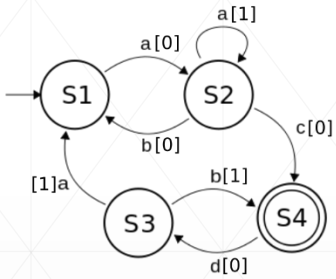
produces

uses

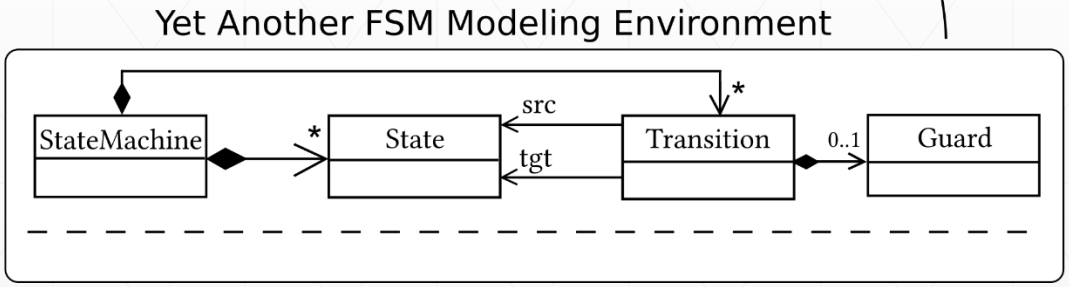
DSL User (Engineer)

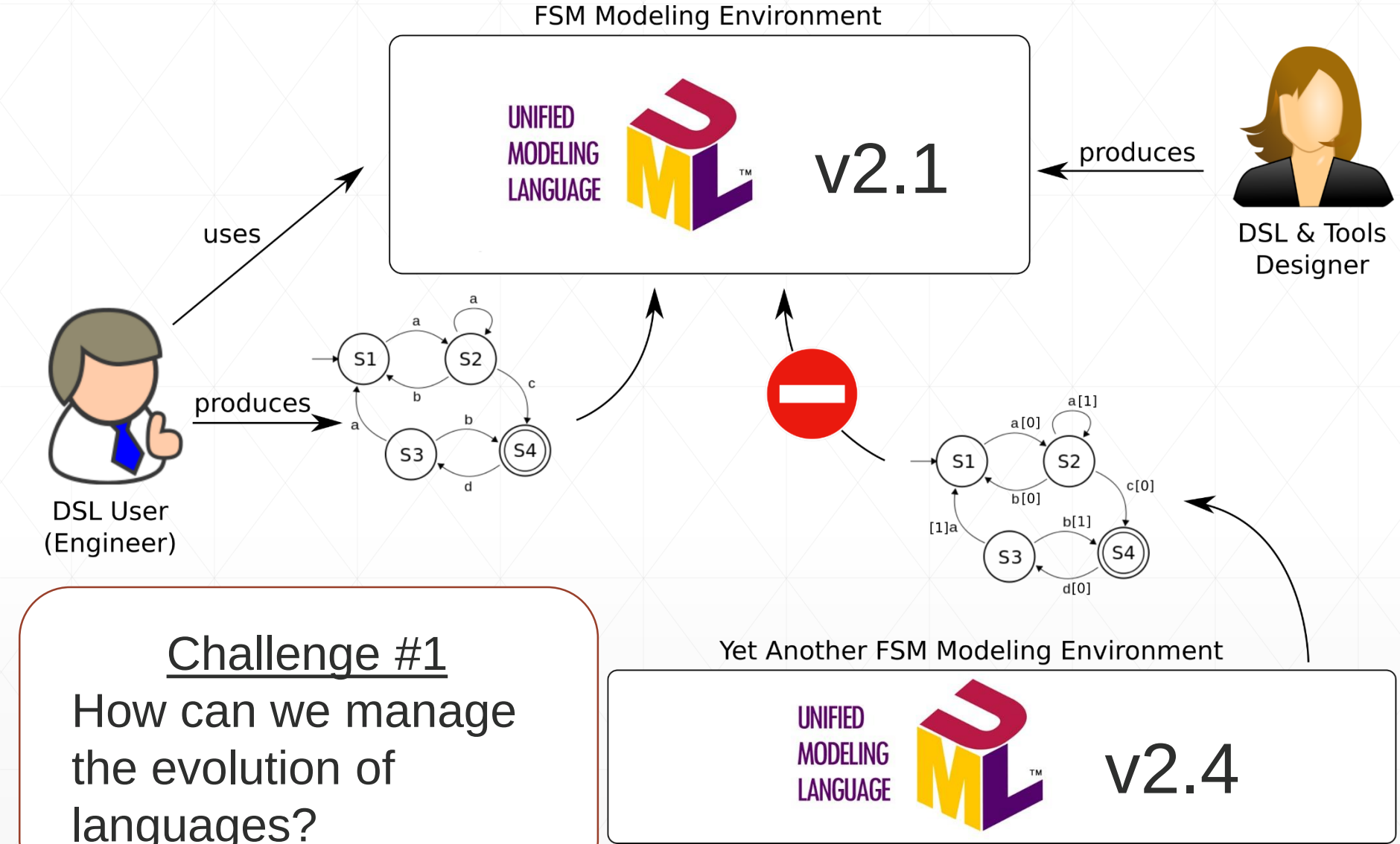
produces





Variants or subsequent versions cannot leverage previous engineering efforts





Challenge #1

How can we manage the evolution of languages?

FSM Modeling Environment

Capella

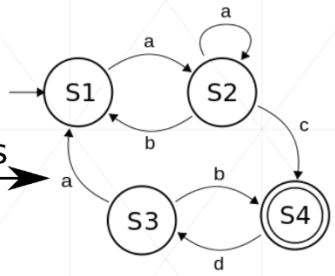
uses

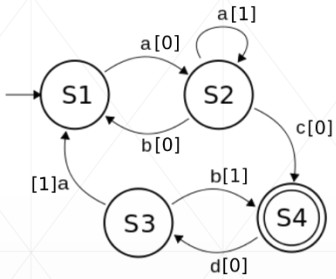
produces

DSL & Tools Designer

DSL User (Engineer)

produces



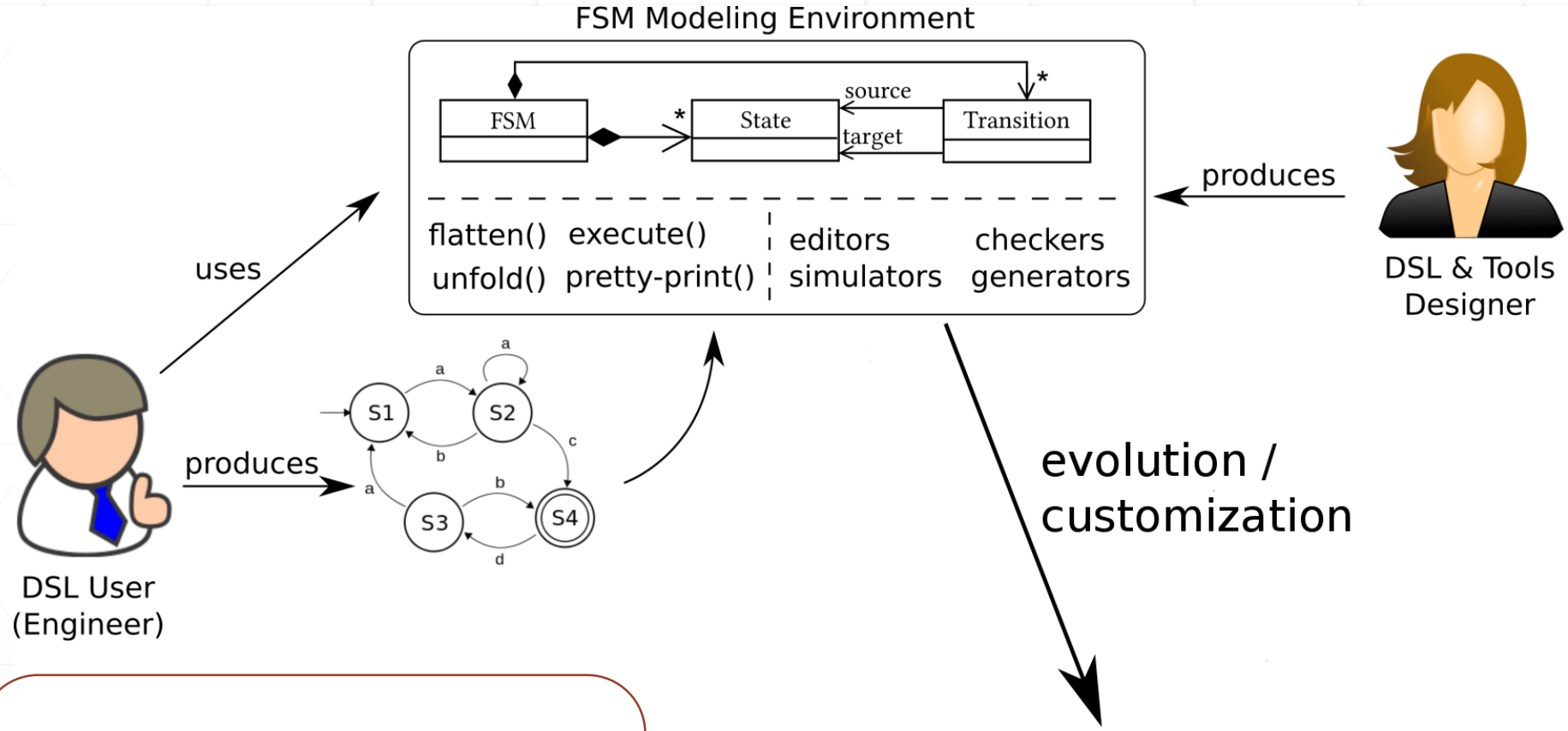


Yet Another FSM Modeling Environment

UNIFIED MODELING LANGUAGE™

Challenge #2

How can we manage the interoperability between similar languages?



Challenge #3

How can we ease the derivation of new, customized languages?

Challenges

DSL & Tools Designer

- Evolve languages
- Manage syntactical / semantical variation points
- Generic tools & transformations
- Reuse / extend / customize existing languages

DSL User

- Agile modeling
- Manipulate models in different environments
- Reuse transformations & tools

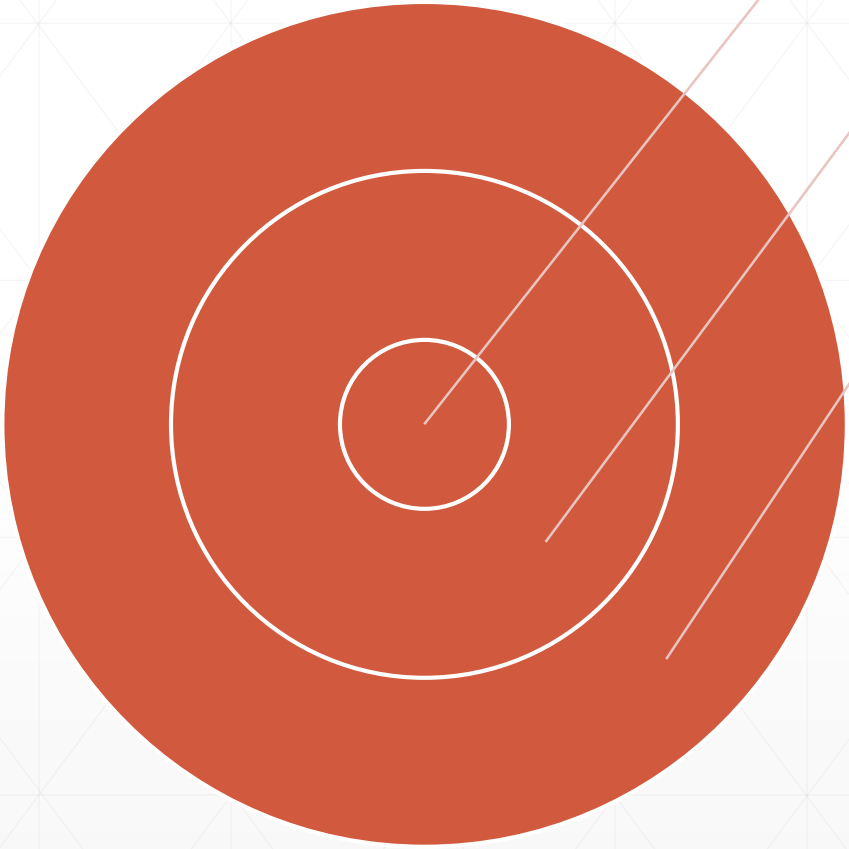
Language Interfaces

A Typing Theory for Software Language Engineering

Language Interfaces

- Tools, transformations, environments are tightly coupled with the language they were originately defined on
  - → If the language evolves, associated tools break
  - → If a variant exists, tools cannot be reused

- An abstraction layer would reduce the coupling
  - → We realize this abstraction layer with language interfaces



Language Implementation

(abstract syntax, concrete syntaxes, semantics, …)

Language Interface

(meaningful information for a specific purpose
In an appropriate formalism)

Language and Model engineering

(transformations, tools, editors, IDEs, …)

→ Multiple DSLs can match the same interface
→ Operators defined on an interface can be reused for all implementing DSLs

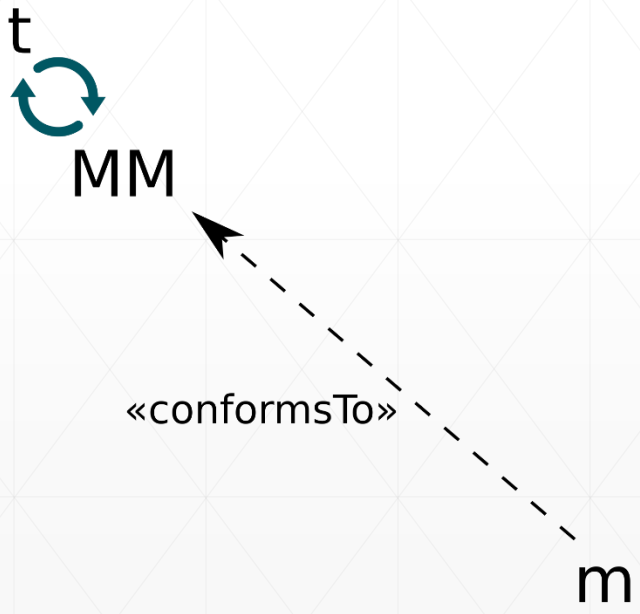
A structural interface: the model type

- Interface over the abstract syntax of a language (a metamodel)
- Focus on the reuse of tools and transformations
- Typing semantics for languages and models

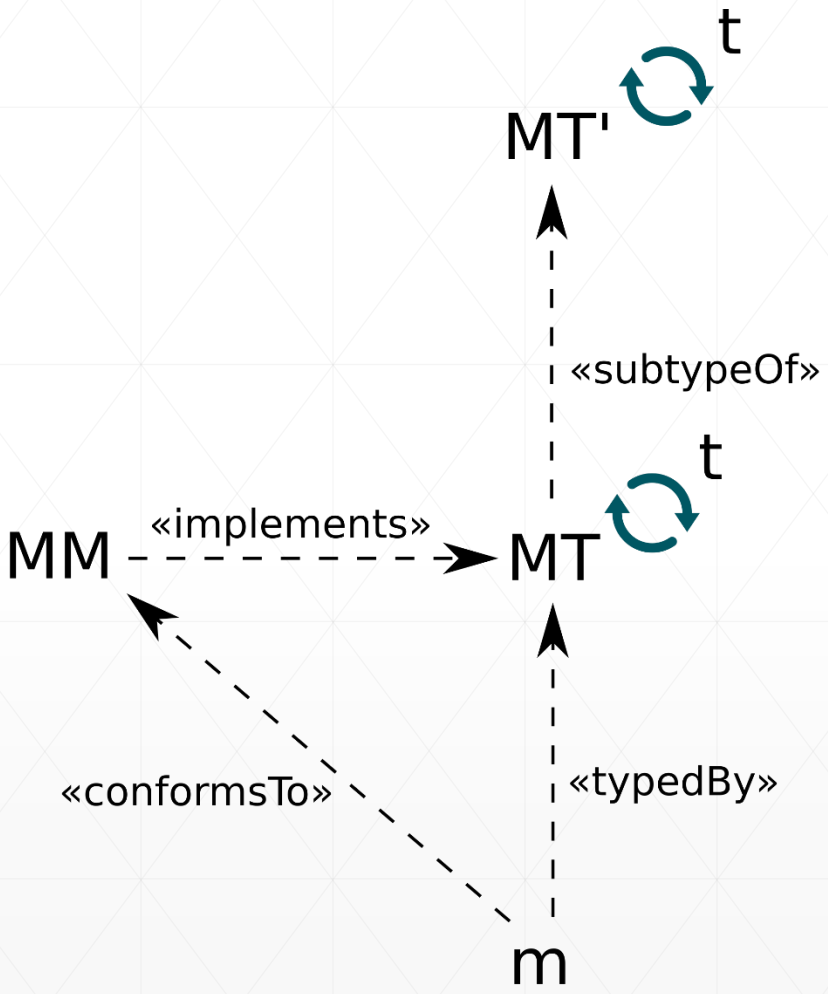
- Supported by a model-oriented type system
- Models (i.e. graph of objects) as first-class citizens
  - → Type group (family) polymorphism
  - → Type groups consistency
  - → Structural typing

→ Provides model polymorphism and substitutability

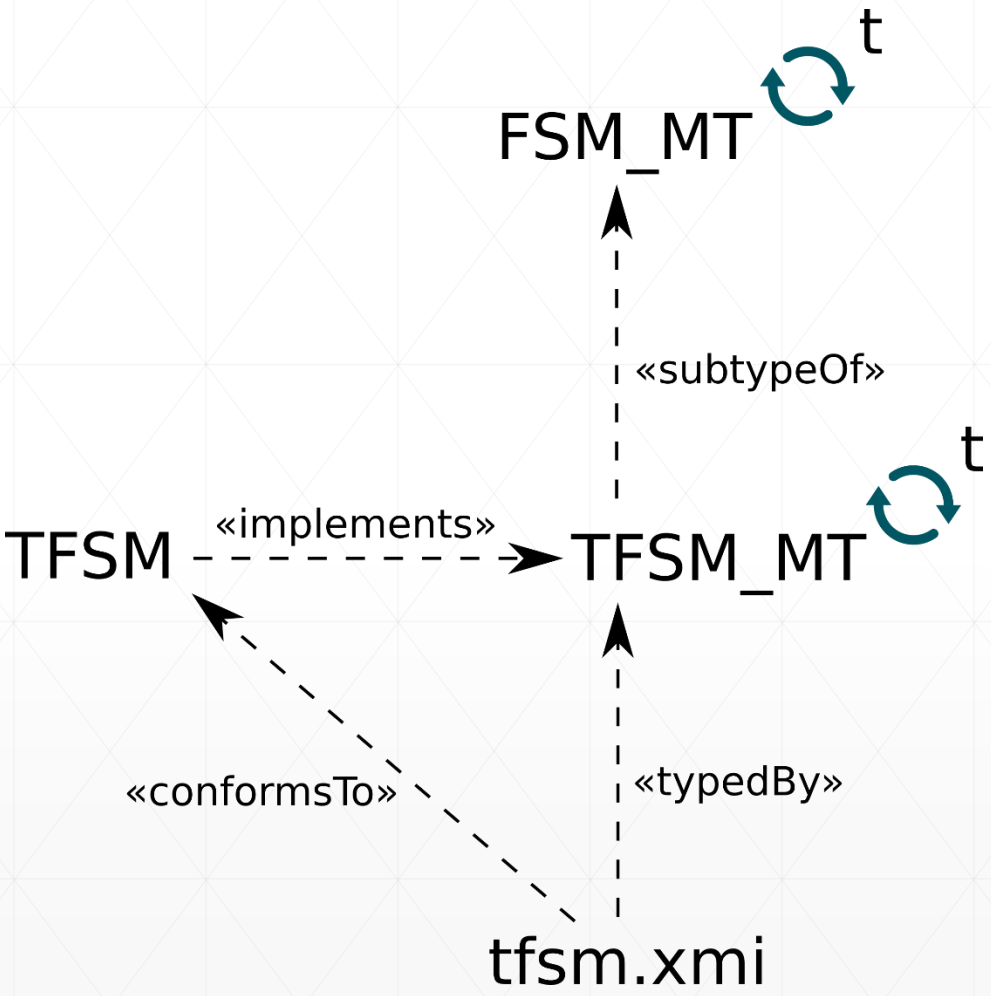
Model typing



Model typing



Model typing



An Algebra for Language Design & Manipulation

Motivations

- DSLs are constantly being developed by independent groups of people
- Given their development costs, leveraging previous engineering efforts is of primary importance

- But
  - Reuse is most of the time unforeseen (legacy / black-box)
  - Imported artifacts may not fit exactly the end-user's requirements

  ➔ must be customized for specialized contexts and environments

An Algebra for Language Manipulation

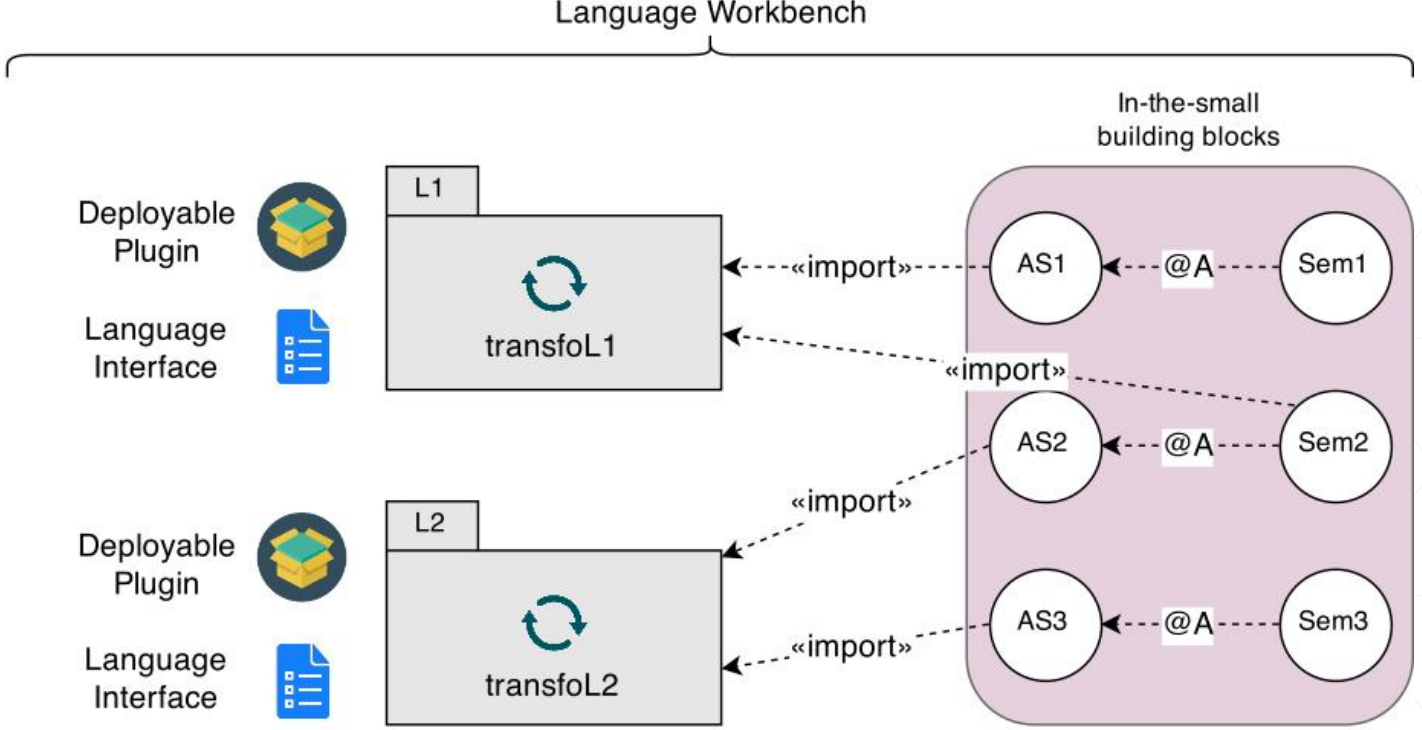
- Fundamental set of operators for
  - Importing (part of) languages
  - Safely assembling them (i.e. statically checking the assembly)
  - Extending pre-existing languages
  - Restricting the scope of languages
  - Specialize/customize them for unforeseen environments

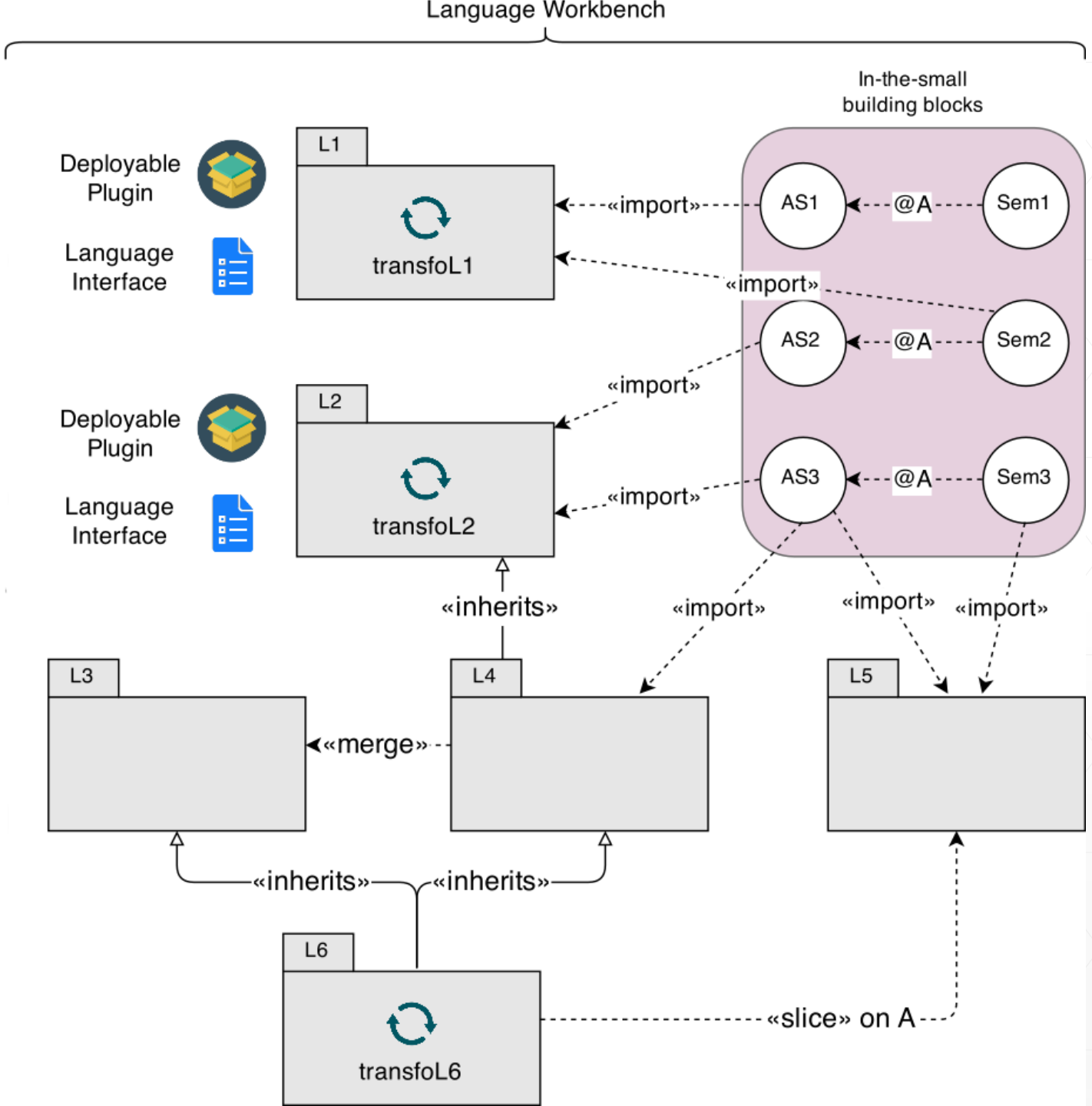
- Implementation of the algebra: a meta-language for safely assembling language artifacts. Results of the assembly are validated, ready for production and reusable for further customization.

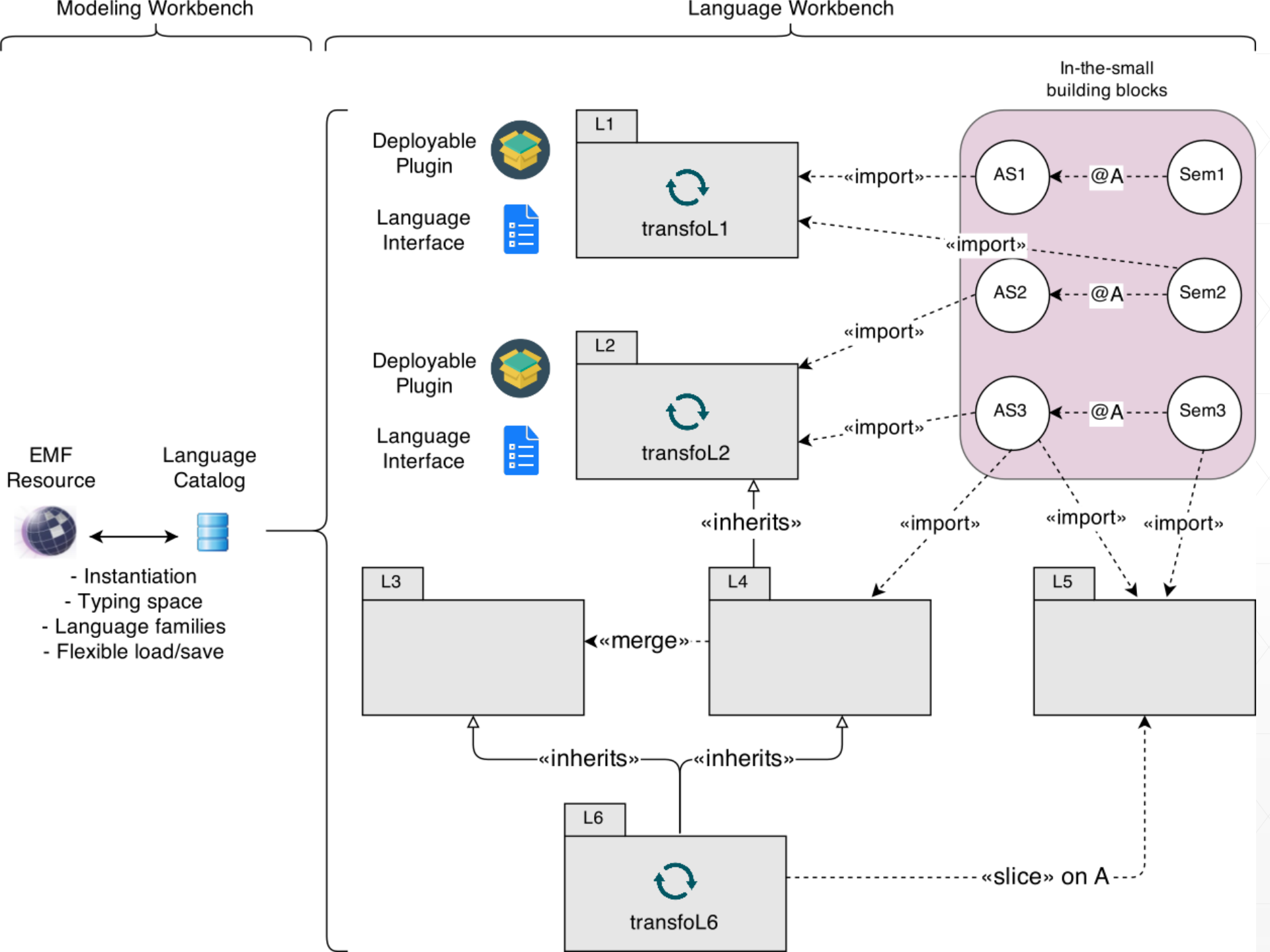
An Algebra for Language Manipulation

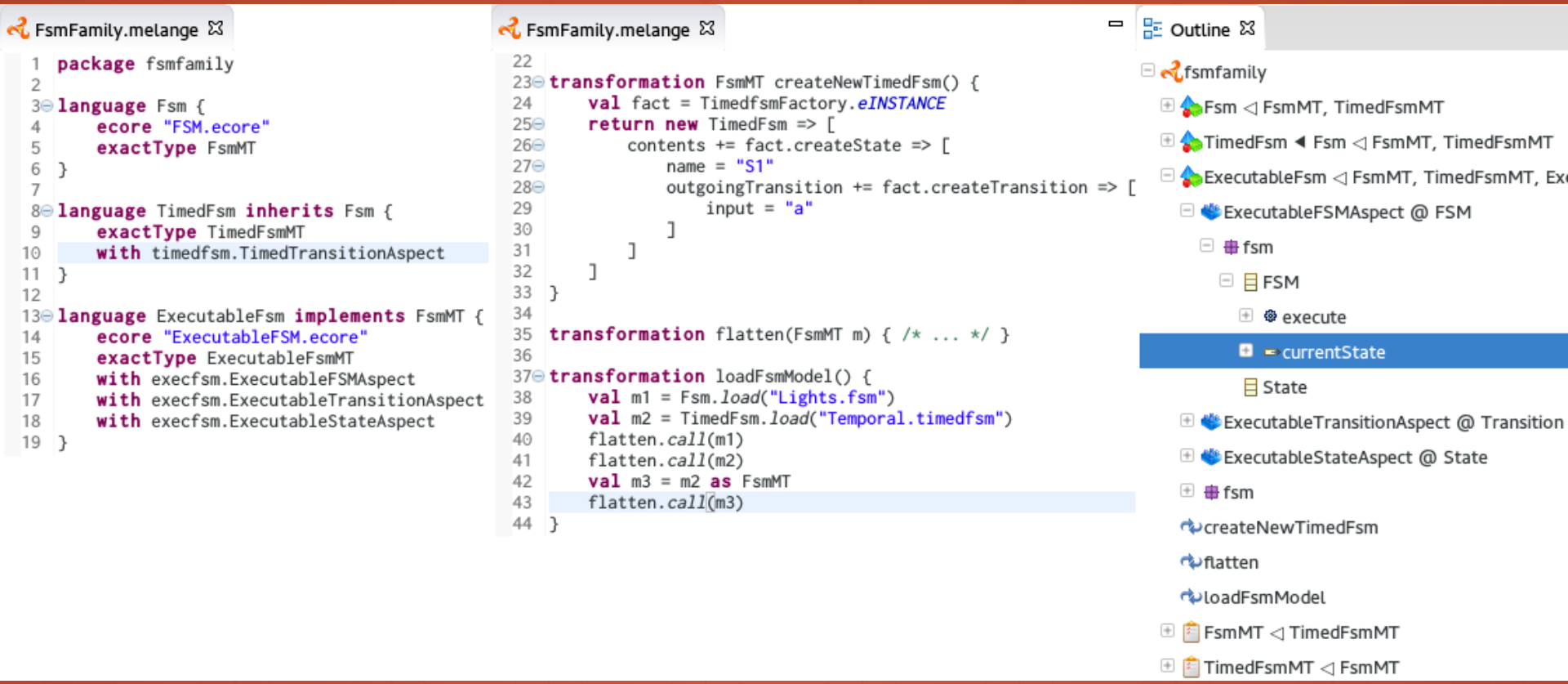
- Language extension: *inheritance* (preserves subtyping!)
- Language restriction: *slice* (given a slicing criterion)
- Language unification: *merge*

- The algebra operates on both syntax and semantics
- Operators are freely composable
- Conflicts management, linearization/disambiguation when appropriate


Language Workbench
In-the-small
building blocks
Deployable
Plugin
Language
Interface
L1
transfoL1
«import»
AS1
@A
Sem1
«import»
AS2
@A
Sem2
Deployable
Plugin
Language
Interface
L2
transfoL2
«import»
«import»
AS3
@A
Sem3


Language Workbench
In-the-small
building blocks
Deployable
Plugin
Language
Interface
L1
transfoL1
«import»
AS1
@A
Sem1
«import»
AS2
@A
Sem2
«import»
Deployable
Plugin
Language
Interface
L2
transfoL2
«import»
AS3
@A
Sem3
«inherits»
«import»
«import»
«import»
L3
«merge»
L4
L5
«inherits»
«inherits»
L6
transfoL6
«slice» on A


Modeling Workbench
Language Workbench
In-the-small
building blocks
Deployable
Plugin
Language
Interface
L1
transfoL1
«import»
AS1
@A
Sem1
«import»
AS2
@A
Sem2
«import»
Deployable
Plugin
Language
Interface
L2
transfoL2
«import»
AS3
@A
Sem3
EMF
Resource
Language
Catalog
- Instantiation
- Typing space
- Language families
- Flexible load/save
«inherits»
«import»
«import»
«import»
L3
«merge»
L4
L5
«inherits»
«inherits»
L6
transfoL6
«slice» on A



Melange

A Language-Based Model-Oriented Programming Language

Melange: A Language-Based Model-Oriented Programming Language

- A language for defining DSLs
  - Assemble pre-existing DSLs building blocks
  - Handy operators for SLE: inheritance, merge, slice, etc.
  - Aspect-oriented modeling (e.g. for executable meta-modeling)
  - Generic transformations

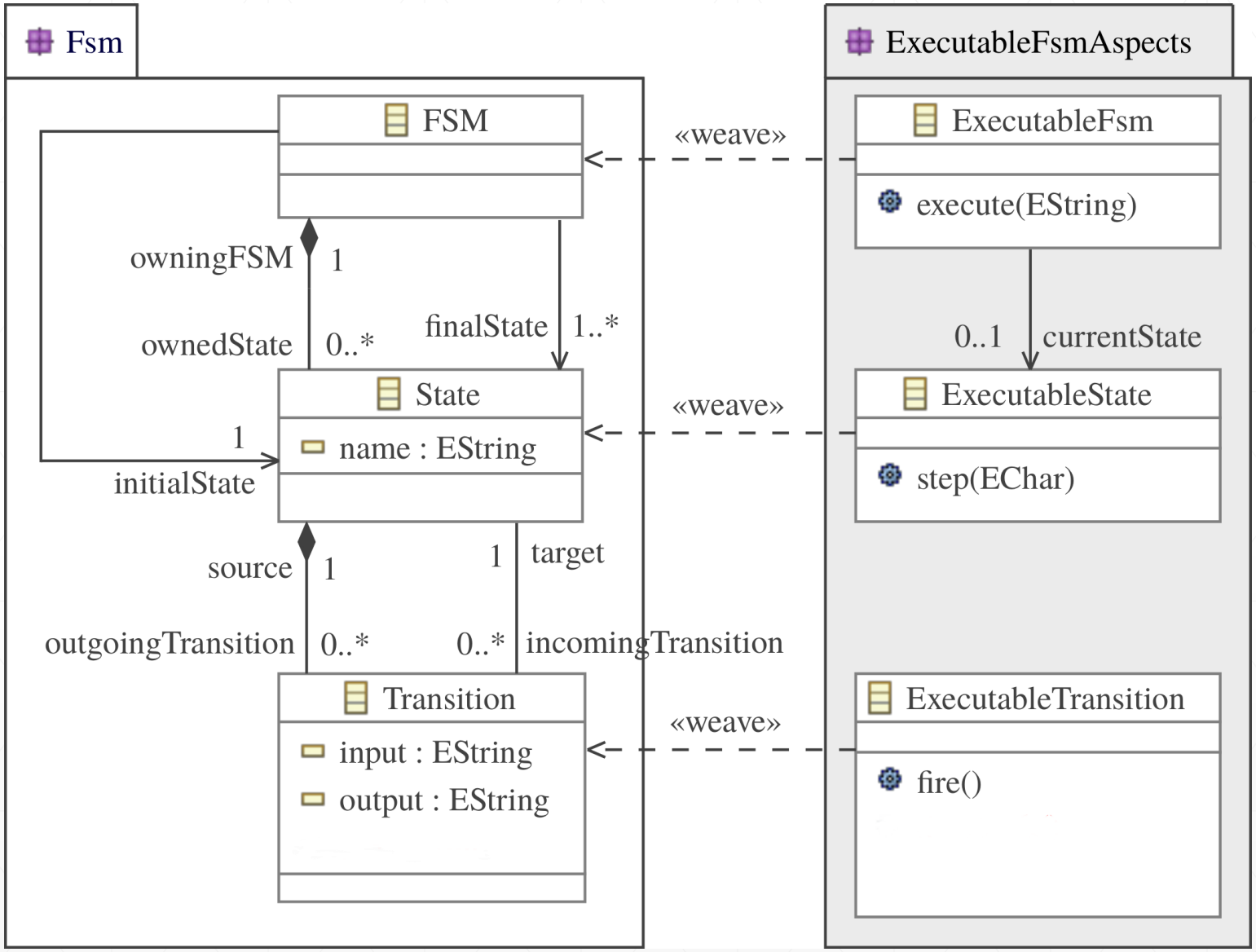
DSL & Tools Designer

- A language for manipulating models
  - Models as first-class, typed citizens
  - Model-oriented type system
  - Providing model polymorphism and substitutability
  - Flexible save and load mechanism

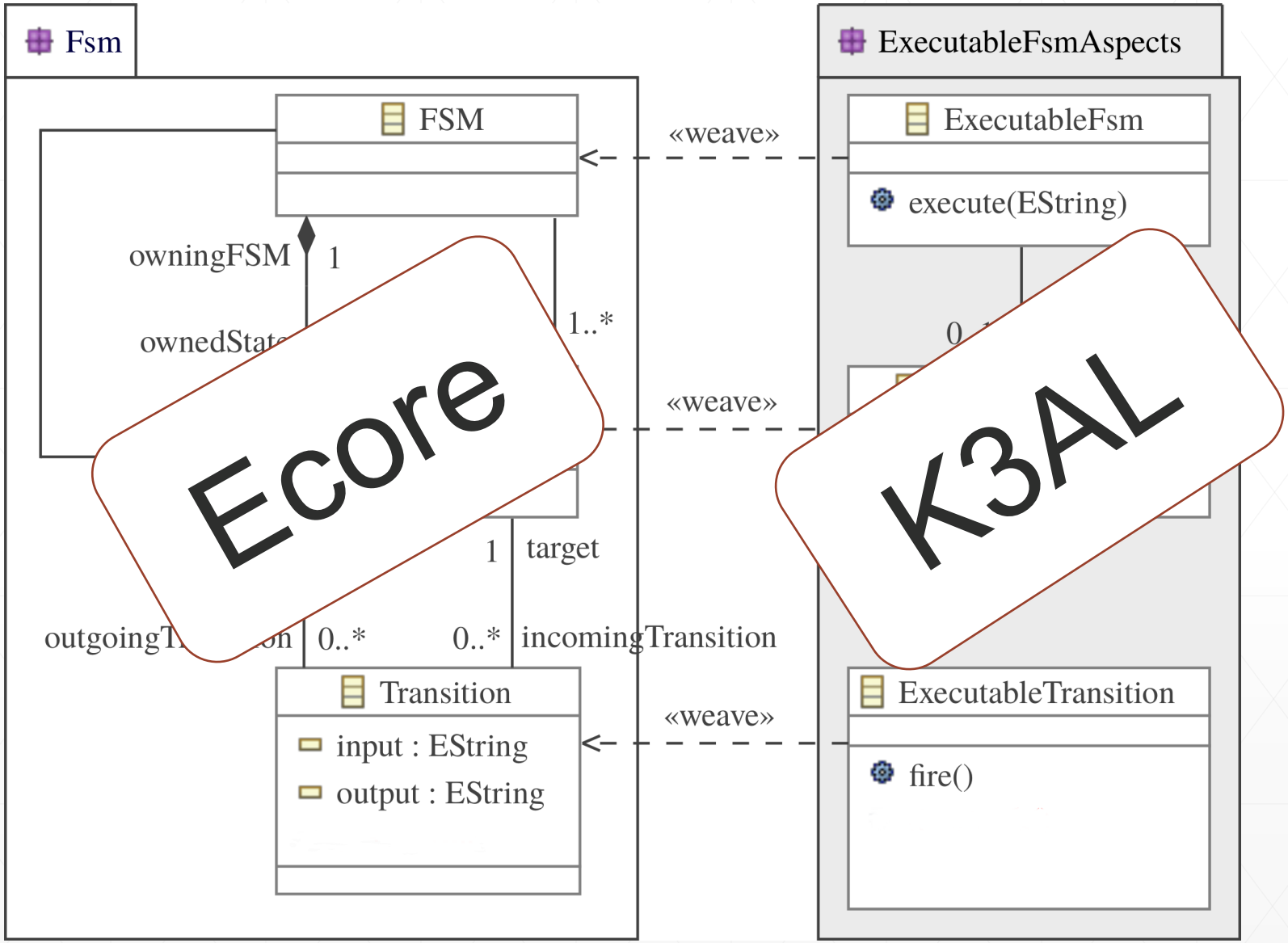
DSL User

- Seamlessly integrated with the EMF ecosystem

DSL Definition in Melange



Dedicated Metalanguages



```
@Aspect(className = FSM)
class FsmAspect {
    State currentState

    def void execute(String s) {
        _self.currentState = _self.initialState
        // read current character, etc.
    }
}

@Aspect(className = State)
class StateAspect {
    def void step(char c) {
        _self.outgoingTransitions.findFirst[input == c].fire
    }
}

@Aspect(className = Transition)
class TransitionAspect {
    def void fire() {
        // fire transition, update current state, etc.
    }
}
```

```
modeltype FsmMT {
    syntax SimpleFSM.ecore
}
```

```
modeltype FsmMT {
    syntax SimpleFSM.ecore
}

language Fsm implements FsmMT {
    syntax FSM.ecore
}
```

```
modeltype FsmMT {
    syntax SimpleFSM.ecore
}

language Fsm implements FsmMT {
    syntax FSM.ecore
}

language ExecFsm {
    syntax FSM.ecore
    with ExecutableSM
    with ExecutableState
    with ExecutableTransition
    exactType ExecFsmMT
}
```

```
modeltype FsmMT {
    syntax SimpleFSM.ecore
}

language Fsm implements FsmMT {
    syntax FSM.ecore
}

language ExecFsm {
    syntax FSM.ecore
    with ExecutableSM
    with ExecutableState
    with ExecutableTransition
    exactType ExecFsmMT
}

language TimedFsm inherits ExecFsm {
    // Variation point
    with TimedTransition
    exactType TimedFsmMT
}
```

```
modeltype FsmMT {
    syntax SimpleFSM.ecore
}

language Fsm implements FsmMT {
    syntax FSM.ecore
}

language ExecFsm {
    syntax FSM.ecore
    with ExecutableSM
    with ExecutableState
    with ExecutableTransition
    exactType ExecFsmMT
}

language TimedFsm inherits ExecFsm {
    // Variation point
    with TimedTransition
    exactType TimedFsmMT
}
```

```
transformation flatten(FsmMT m) {
    m.root.ownedStates.forEach[...]
}
```

```
modeltype FsmMT {
    syntax SimpleFSM.ecore
}

language Fsm implements FsmMT {
    syntax FSM.ecore
}

language ExecFsm {
    syntax FSM.ecore
    with ExecutableSM
    with ExecutableState
    with ExecutableTransition
    exactType ExecFsmMT
}

language TimedFsm inherits ExecFsm {
    // Variation point
    with TimedTransition
    exactType TimedFsmMT
}
```

```
transformation flatten(FsmMT m) {
    m.root.ownedStates.forEach[...]
}

transformation execute(ExecFsmMT m){
    // With dynamic binding
    m.root.execute(«word»)
}
```

```
modeltype FsmMT {
    syntax SimpleFSM.ecore
}

language Fsm implements FsmMT {
    syntax FSM.ecore
}

language ExecFsm {
    syntax FSM.ecore
    with ExecutableSM
    with ExecutableState
    with ExecutableTransition
    exactType ExecFsmMT
}

language TimedFsm inherits ExecFsm {
    // Variation point
    with TimedTransition
    exactType TimedFsmMT
}
```

```
transformation flatten(FsmMT m) {
    m.root.ownedStates.forEach[...]
}

transformation execute(ExecFsmMT m){
    // With dynamic binding
    m.root.execute(«word»)
}

main() {
    val m1 = new Fsm
    val m2 = ExecFsm.load(«Foo.fsm»)
    val m3 = TimedFsm.load(«Foo.tfsm»)

    val m4 = m3 as FsmMT // Viewpoints

    flatten(m1)
    flatten(m2)
    flatten(m3)
    execute(m2)
    execute(m3)
    ~~execute(m1)~~ // Statically forbidden
}
```

```
language CustomizedFsm inherits Fsm {
    merges HierarchicalFsm

    slices Xtend on [XExpression] /* Provides an action language
                                     extracted from Xtend */
    with SpecializedTransition    /* Customized semantics for
                                     Transition::fire */
}

@Aspect(className = Transition)
class SpecializedTransition {
    XExpression guard
    XExpression action

    def void fire() {
        if (guard) {
            action()
            super.fire()
        }
    }
}
```

Ongoing Experiments

- Families of syntactically and semantically diverse languages
  - Example: FSM
    - Syntaxes: Simple – hierarchical – with time constraints – etc.
    - Semantics: Run-to-completion – concurrent – etc.
    - Generic transformations: flatten – execute – etc.
- Thales' Capella language
  - xCapella: executable extension of Capella
  - Maximizing the interoperability with UML
- fUML

Wrap-up

DSL & Tools Designer

- DSL engineering
  - High-level operators
    - Inheritance, merge, etc.
    - Aspect-oriented modeling
  - Executable meta-modeling
  - Generic tools definition

DSL User

- Agile modeling
  - Manipulate models in different environments
  - Viewpoints
  - Reuse of tools

Questions?

http://melange-lang.org
https://github.com/diverse-project/melange