Building Generic Tools for Domain-Specific Languages

IPA Fall Days on Models in Software Engineering

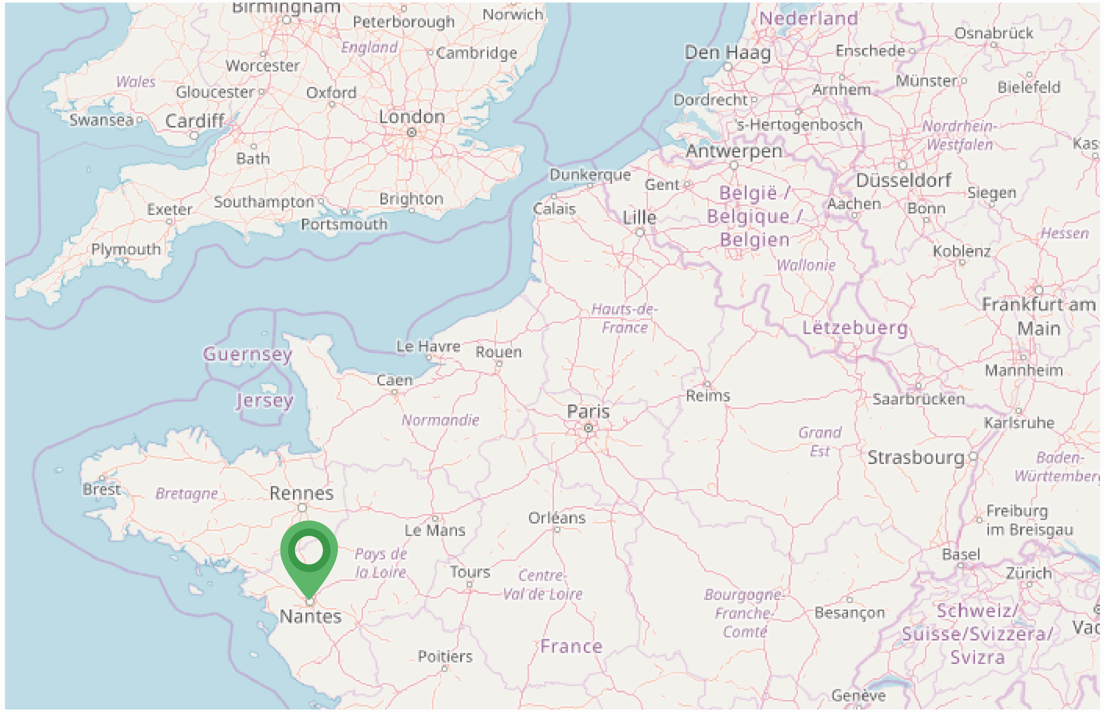
Erwan Bousse (Université de Nantes, *France*)

erwan.bousse@ls2n.fr

October 31st, 2018

About me

Where I come from



Université de Nantes

- 38 000 students
- 21 faculties

LS2N (Lab. of Digital Sciences of Nantes)

- *450 researchers* in a "Joint Research Unit" shared by 5 public institutions (including Univ. Nantes)
- *5 areas of expertise*: "Systems Design and Operation", "Robotics, Processes and Calculation", "Data Science and Decision-making", "Signals, Images, Ergonomics and Languages", "Software and Distributed Systems Science"

UNIVERSITÉ DE NANTES

LS2N

What I do

Currently

- Associate Professor at the Université de Nantes teaching software engineering (including MDE and SLE) at the *Department of Computer Science*
- Member of the NaoMod research group in the LS2N lab, which works on a wide range of MDE topics:
  - model transformation languages (ATL, CoqTL)
  - efficient model storage (NeoEMF)
  - runtime models management
  - scalable model views (EMF Views)
  - *software language engineering, model execution (GEMOC Studio)*

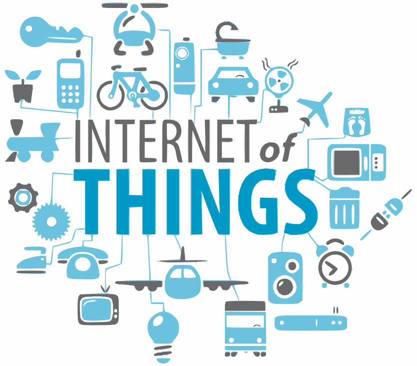
Before

- 2012–2015: PhD at the University of Rennes (France)
- 2016–2018: Post-doc at TU Wien (Austria)

Background: models, executable DSLs and tools

Increasing complexity of systems

AIRBUS A380

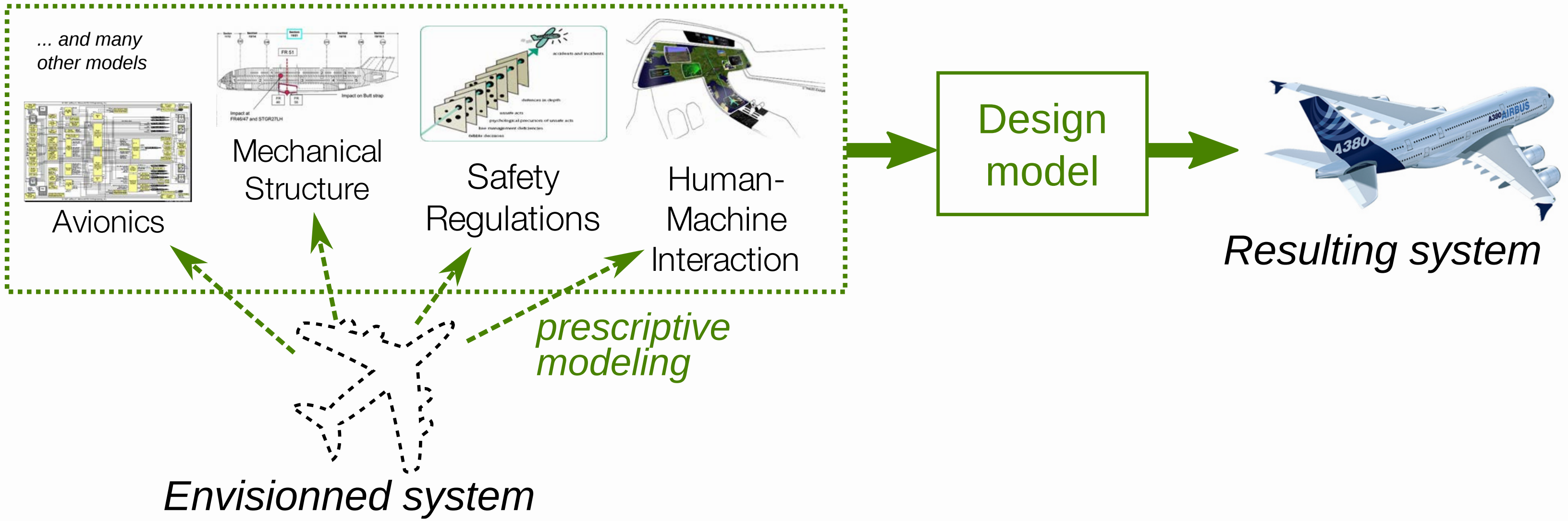


CCP

EVE
ONLINE

- Cyber physical systems, internet of things, massively multiplayer online games, artificial intelligence, ...
  - complexity everywhere!
  - involving multiple stakeholders and concerns from diverse and heterogeneous domains
- Increasing use of software, aka. software–intensive systems

Model-Driven Engineering (MDE)



MDE in a nutshell

1. Separation of concerns through the use of models
   - defined using domain specific languages (DSLs)
   - each representing a particular aspect of a system
2. Composition of all often heterogeneous models
3. Implementation (or generation) of the final resulting system

Domain-Specific Languages (DSLs)

Definition

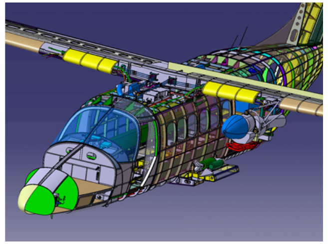
- Well scoped language, often small
- Targets particular tasks in a certain domain
- Relies on dedicated notations (textual or graphical)

Promises

- Less redundancy
- Better separation of concerns
- Accessible for domain experts

```
ComputeDt: ∀n∈ℕ,
δt^{n=0}=option_δt_ini; ,
δt^{n+1}=option_δt_cfl*min{j∈cells}(δtj{j});
```

∇-Nabla (Numerical-analysis)



CATIA (Computed-aided manufacturing)

```html
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

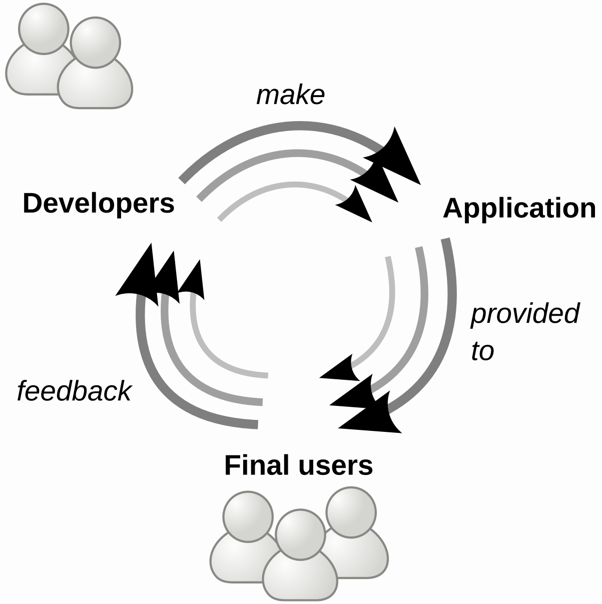
HTML (Web development)

```
background   { color rgb <0.25, 0.25, 0.25> }
camera       { location  <0.0, 0.5, -4.0>
               direction 1.5*z
               right     x*image_width/image_height
               look_at   <0.0, 0.0, 0.0> }
light_source { <0, 0, 0>
               color rgb <1, 1, 1>
               translate <-5, 5, -5> }
```

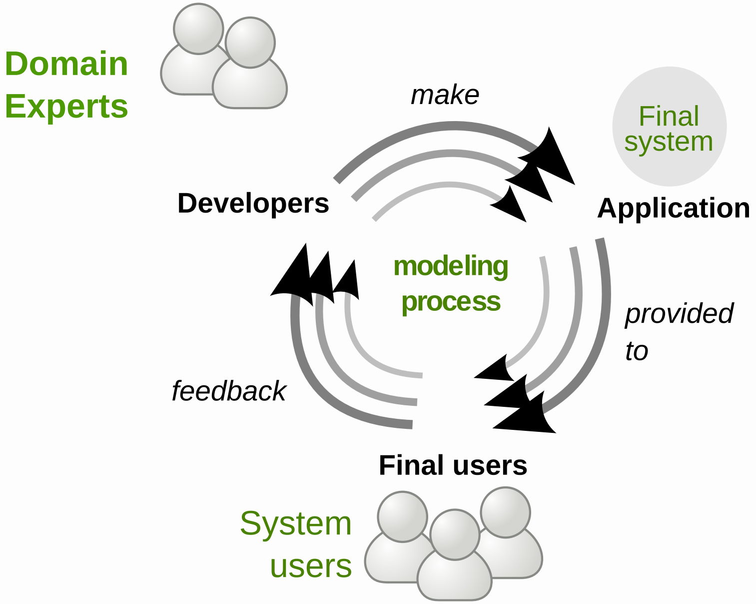
POV-Ray (Computer graphics)

Engineering Domain Specific Languages (DSLs)

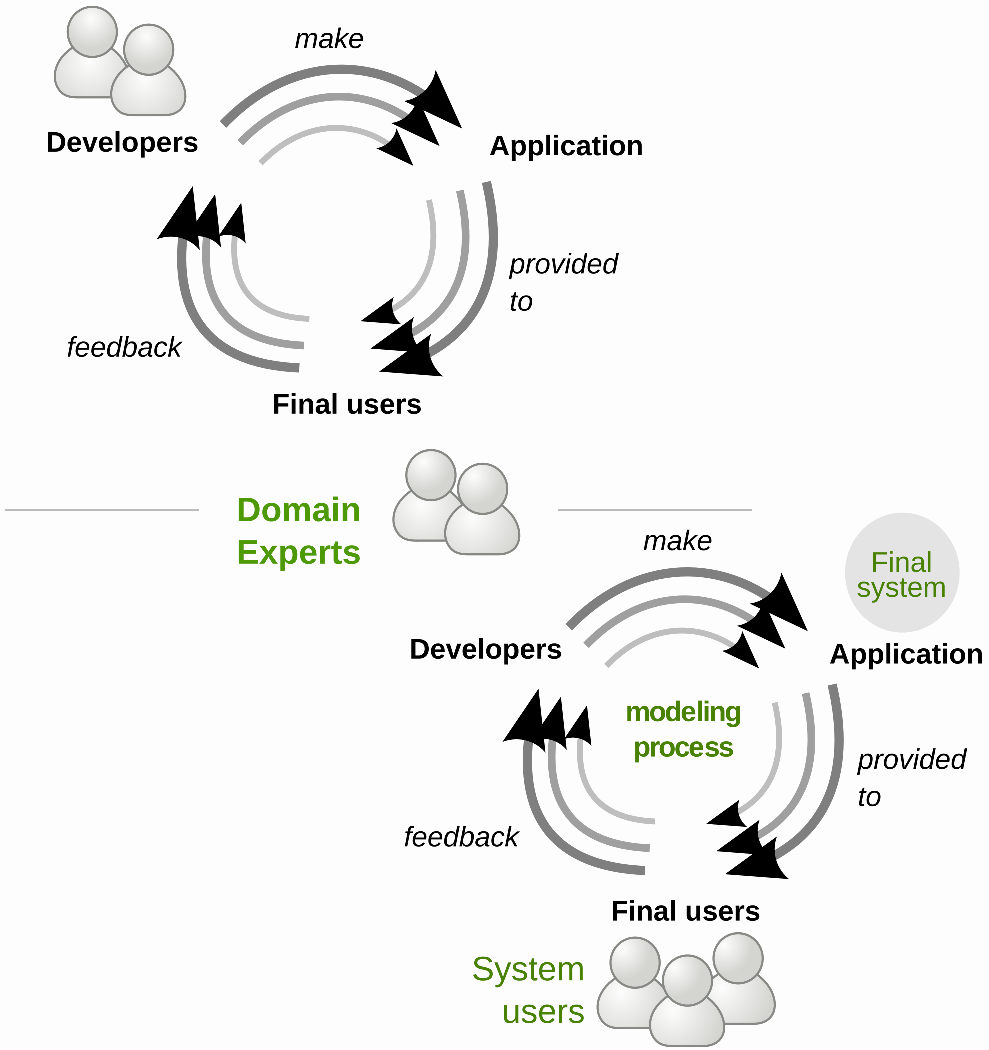
Engineering Domain Specific Languages (DSLs)



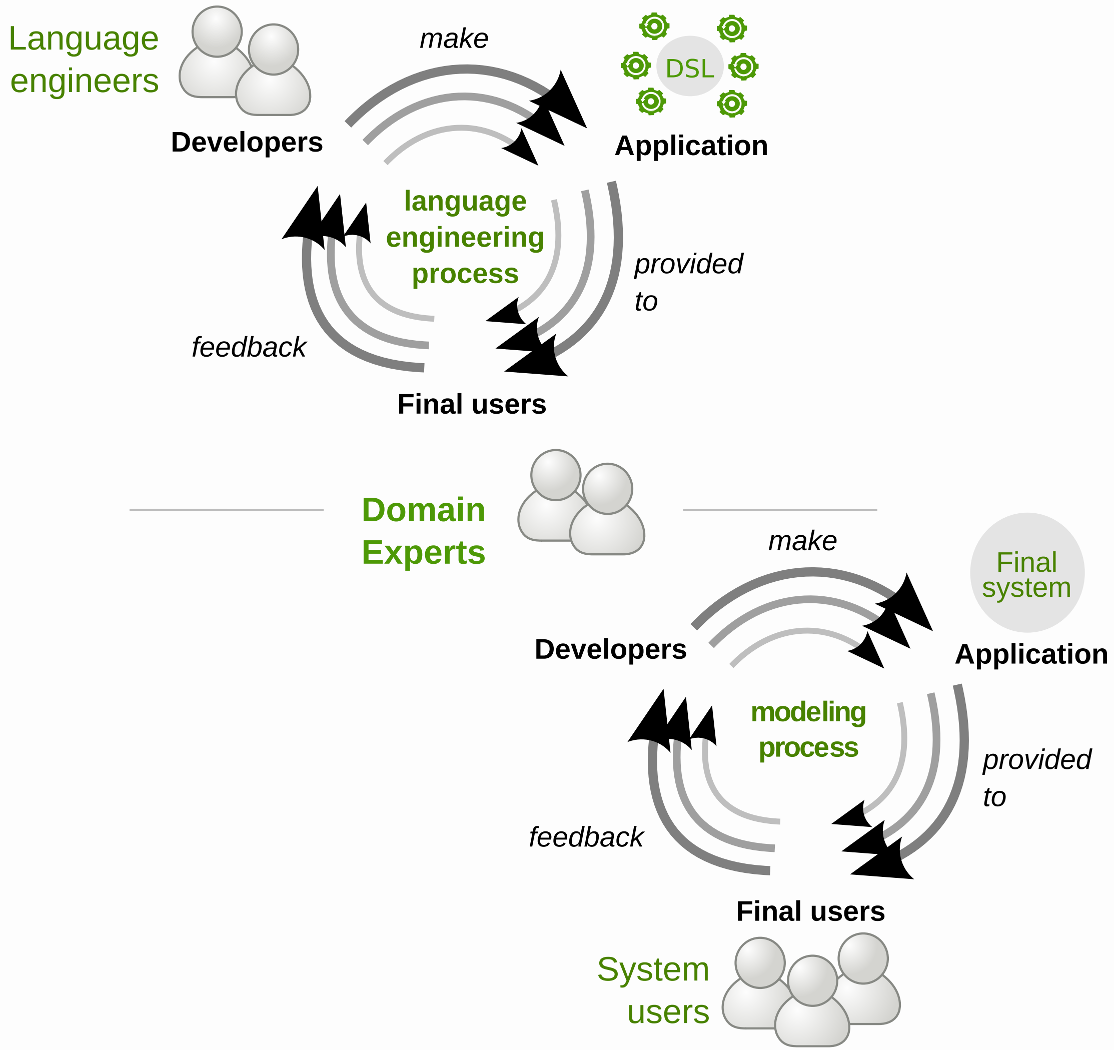
Engineering Domain Specific Languages (DSLs)



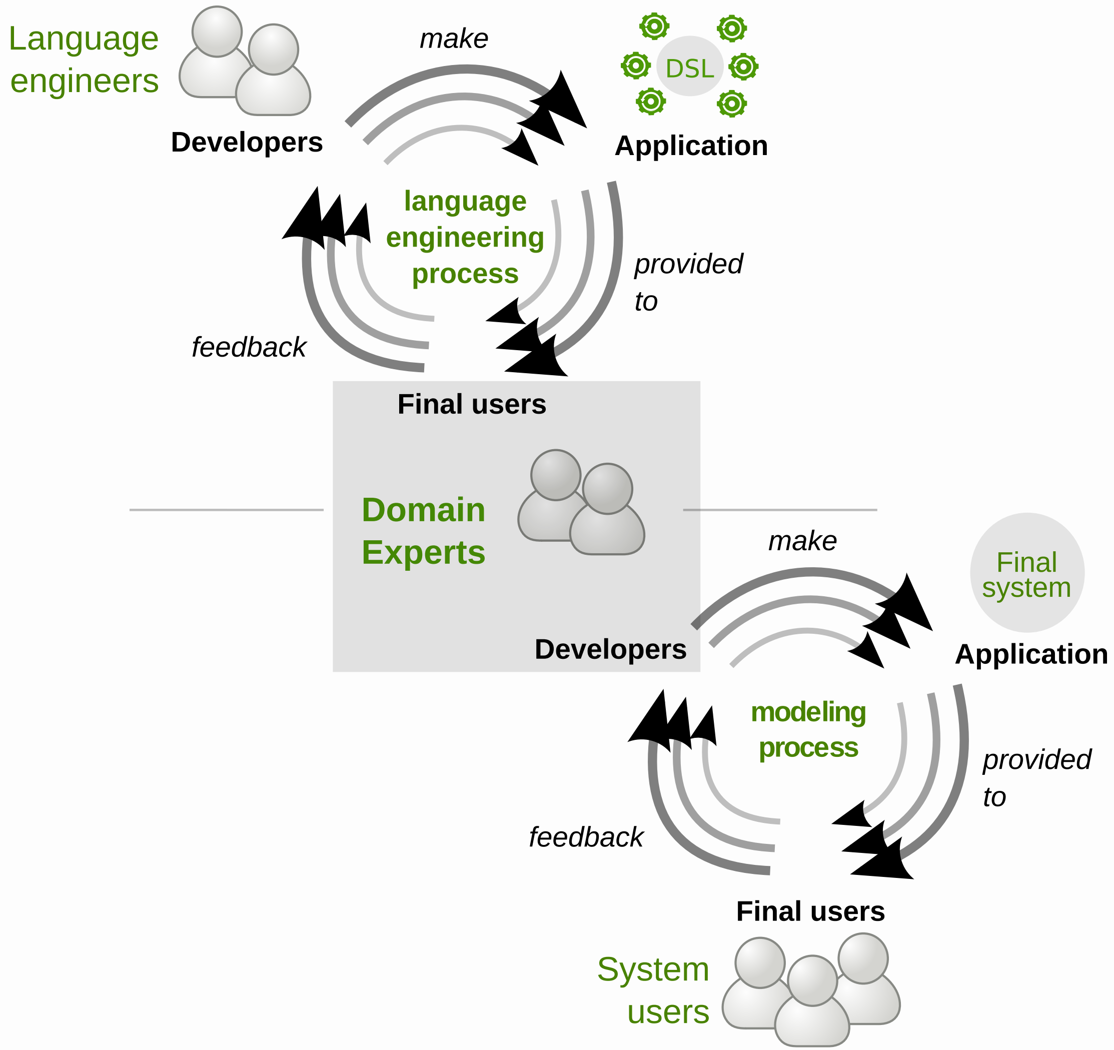
Engineering Domain Specific Languages (DSLs)



Engineering Domain Specific Languages (DSLs)

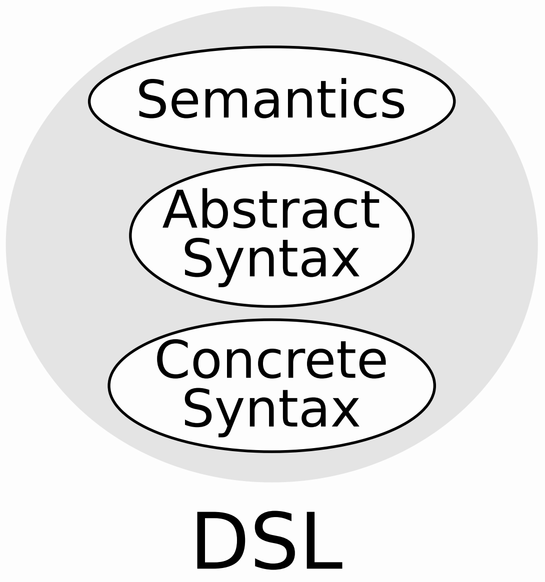


Engineering Domain Specific Languages (DSLs)

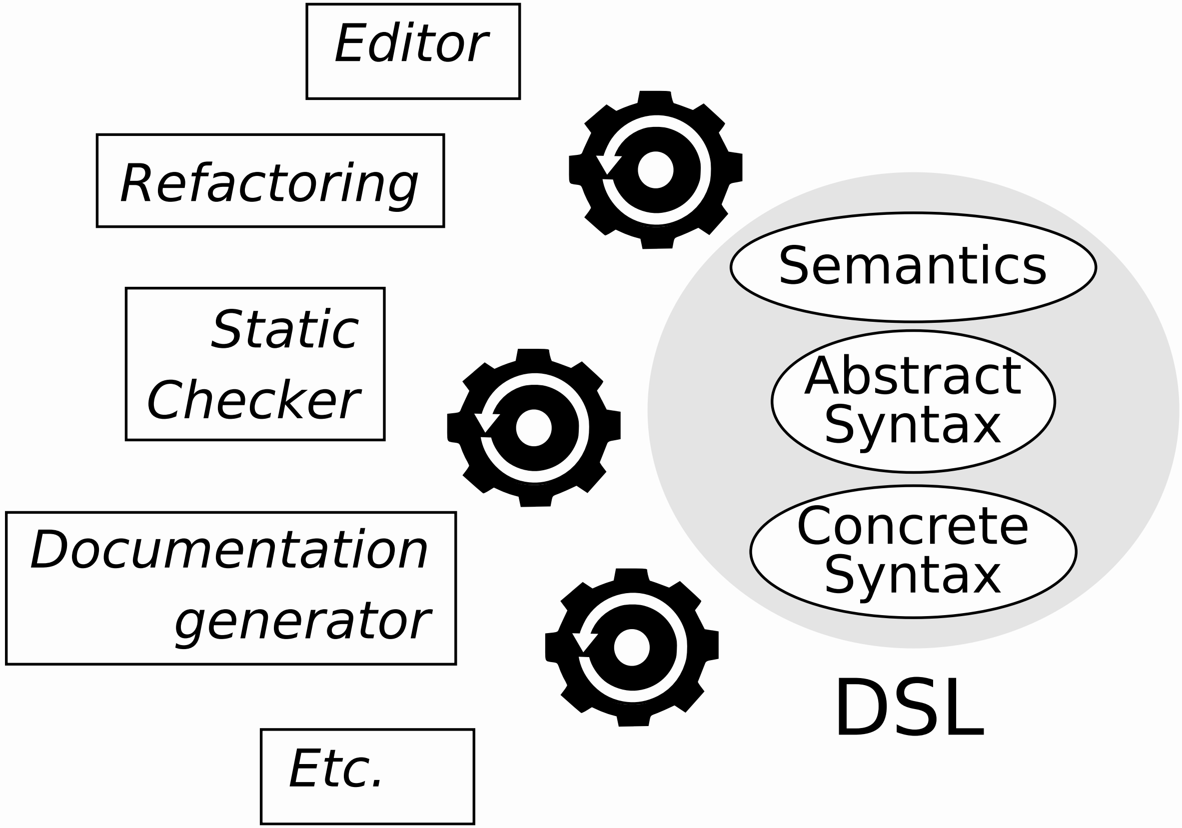


Anatomy and tooling of a DSL

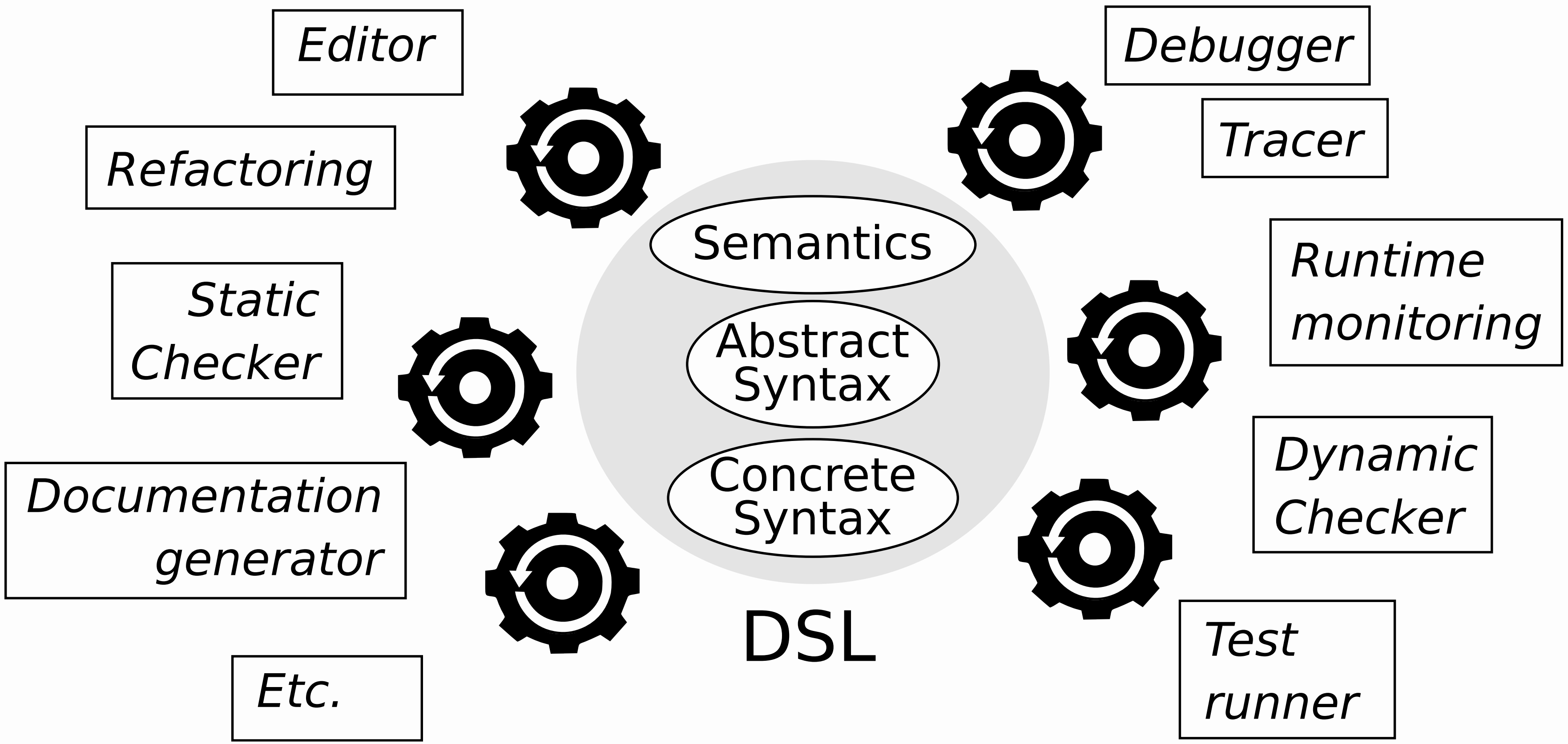
Anatomy and tooling of a DSL



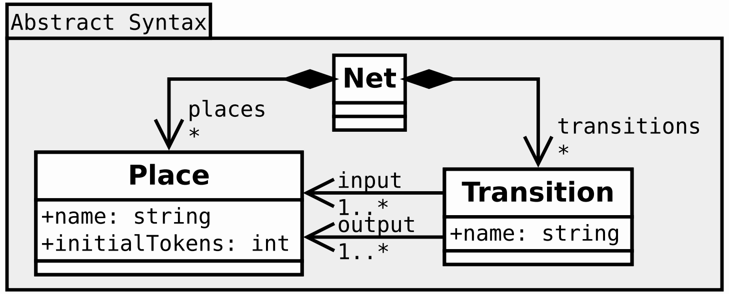
Anatomy and tooling of a DSL



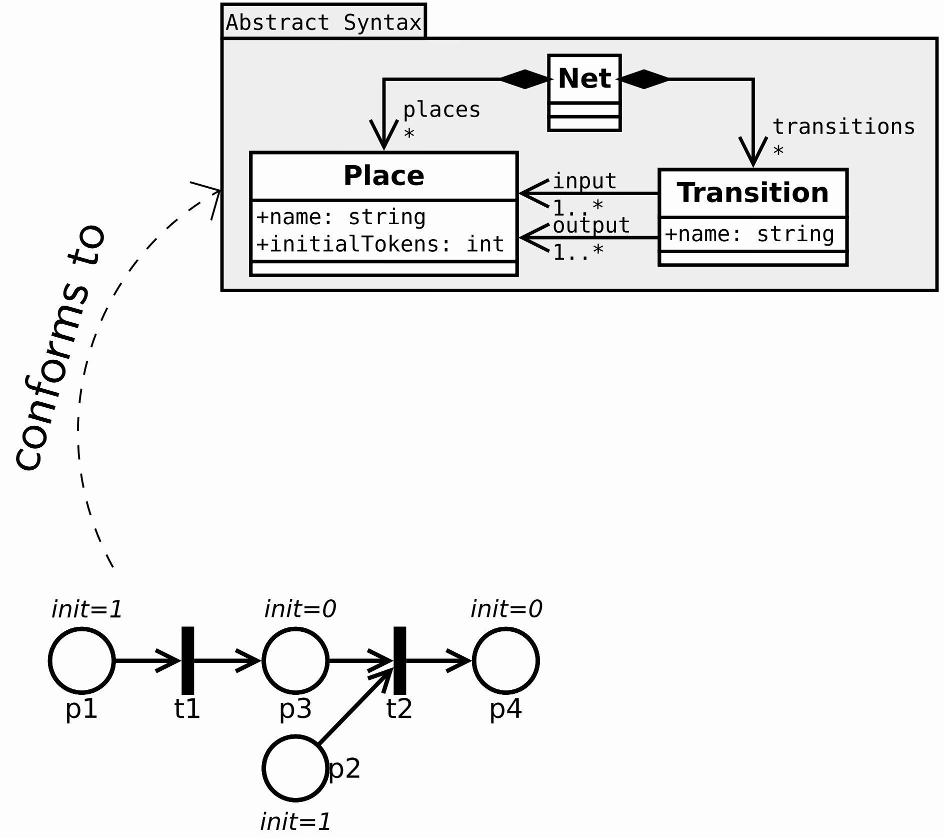
Anatomy and tooling of a DSL



Example of (executable) DSL

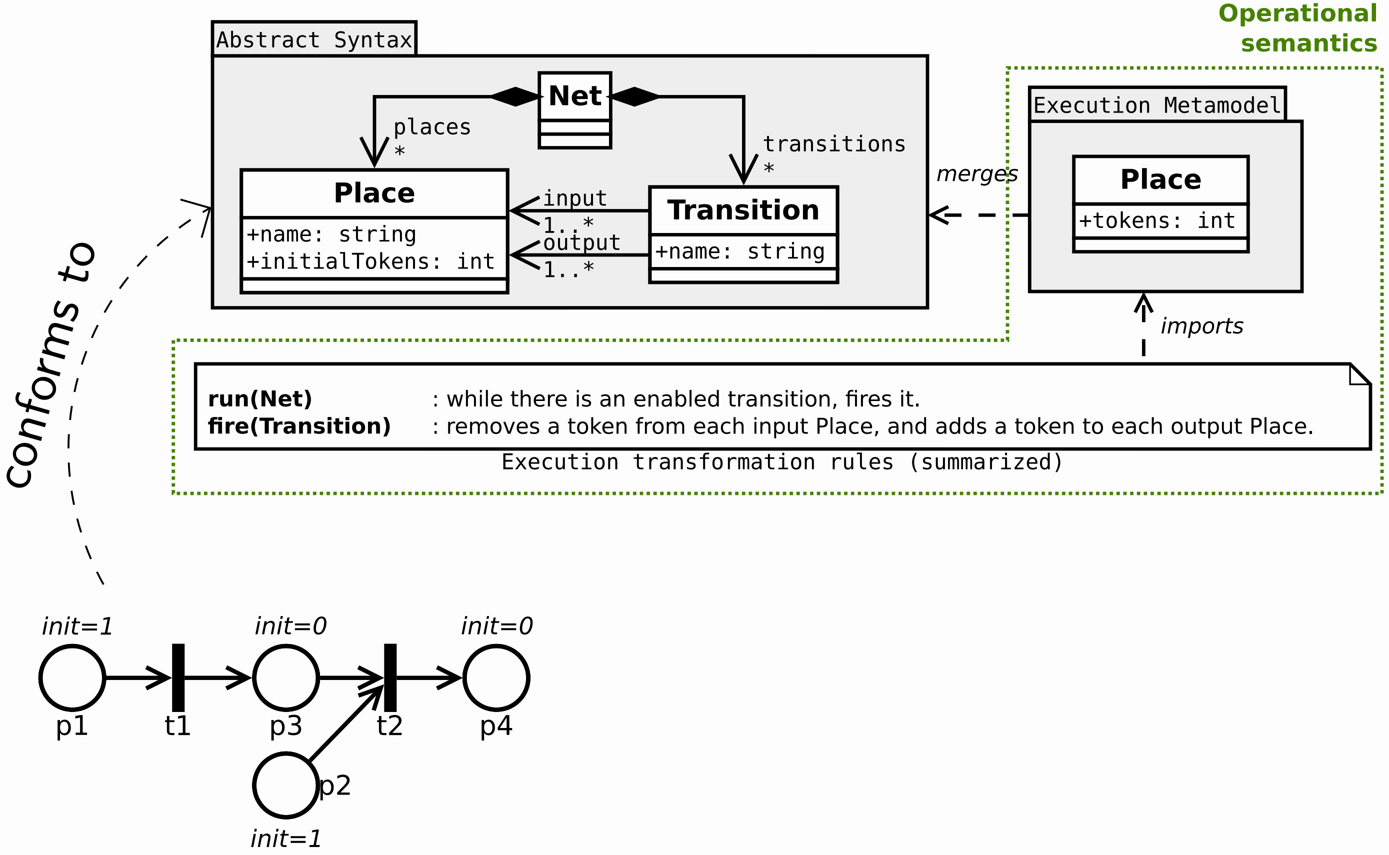


Example of (executable) DSL



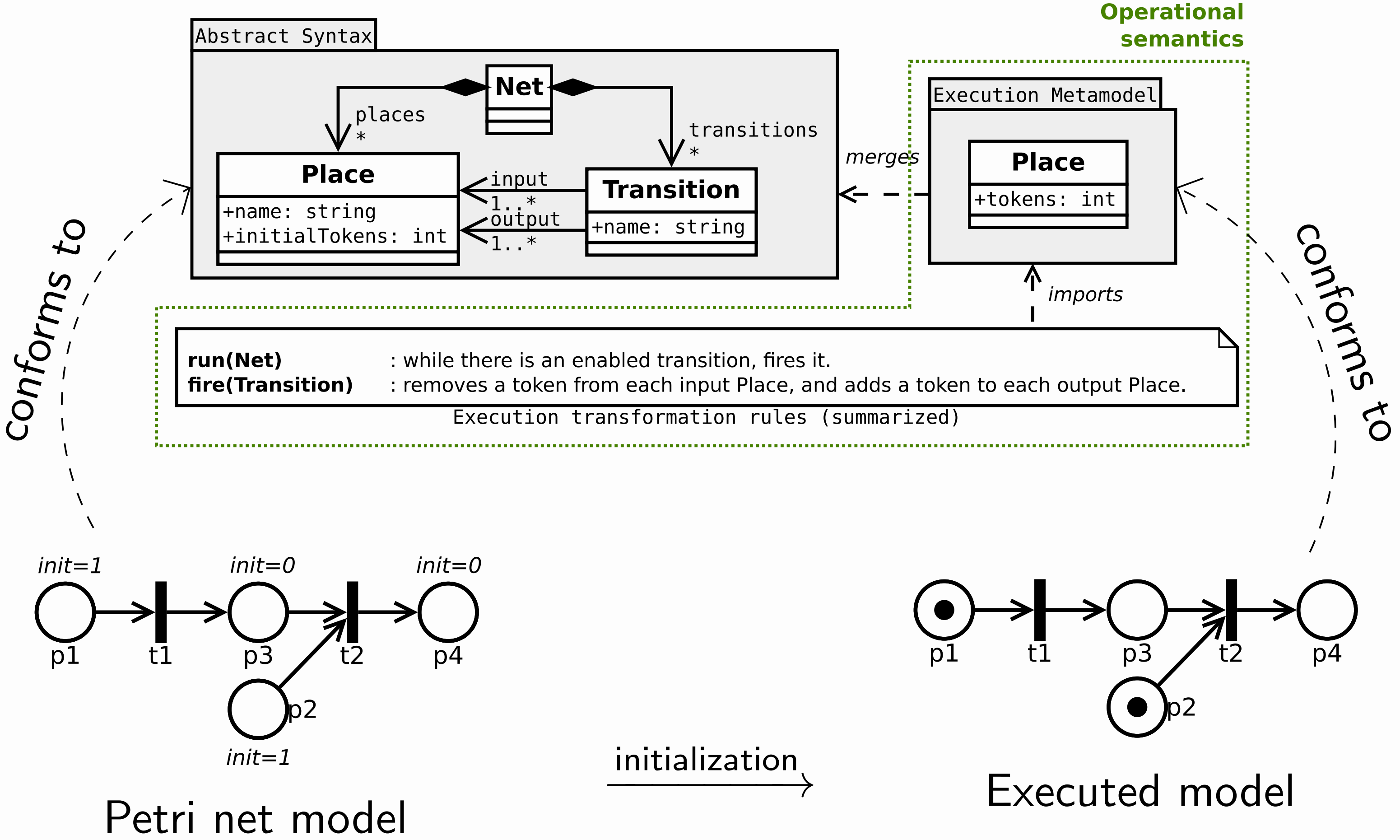
Petri net model

Example of (executable) DSL

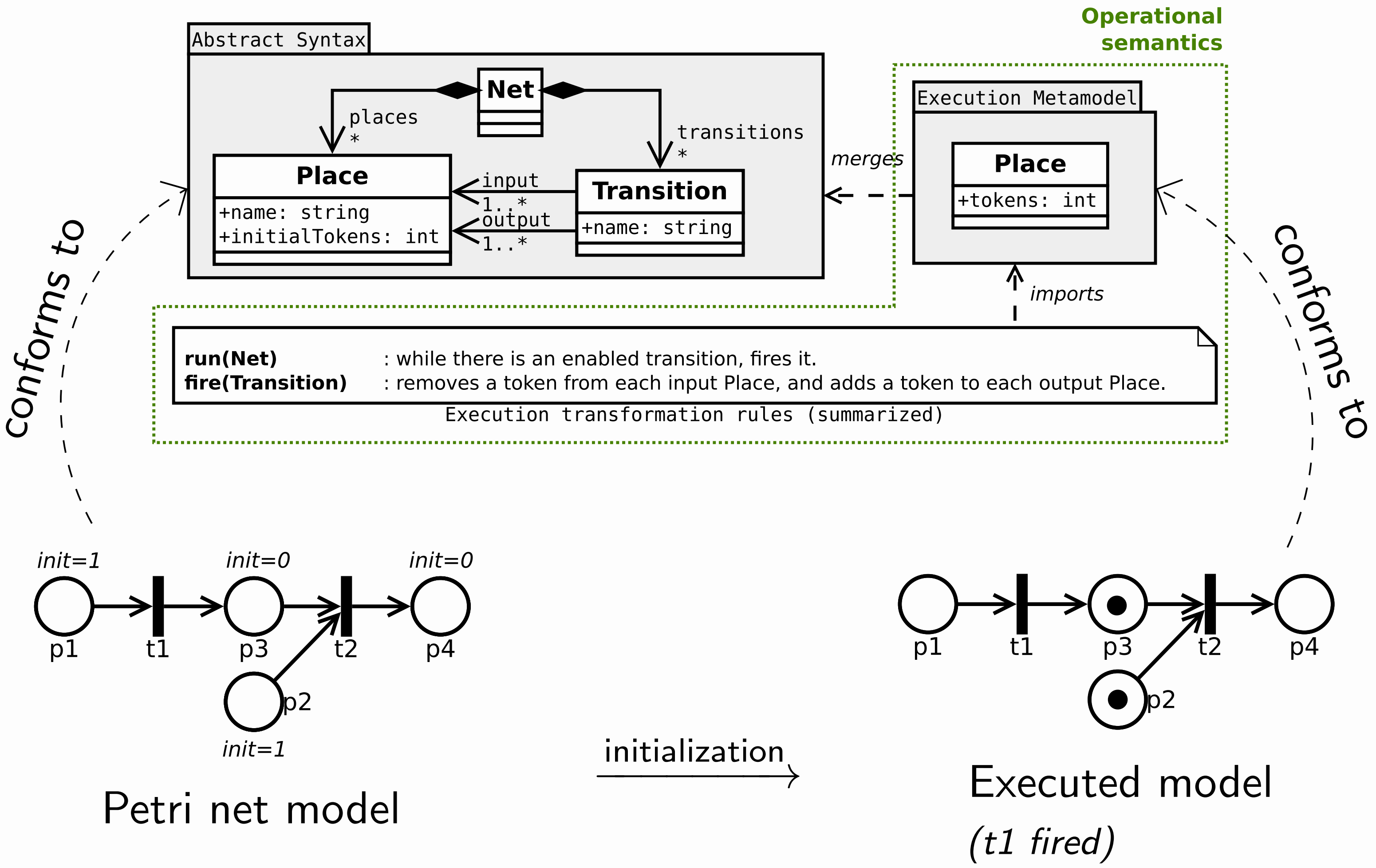


Petri net model

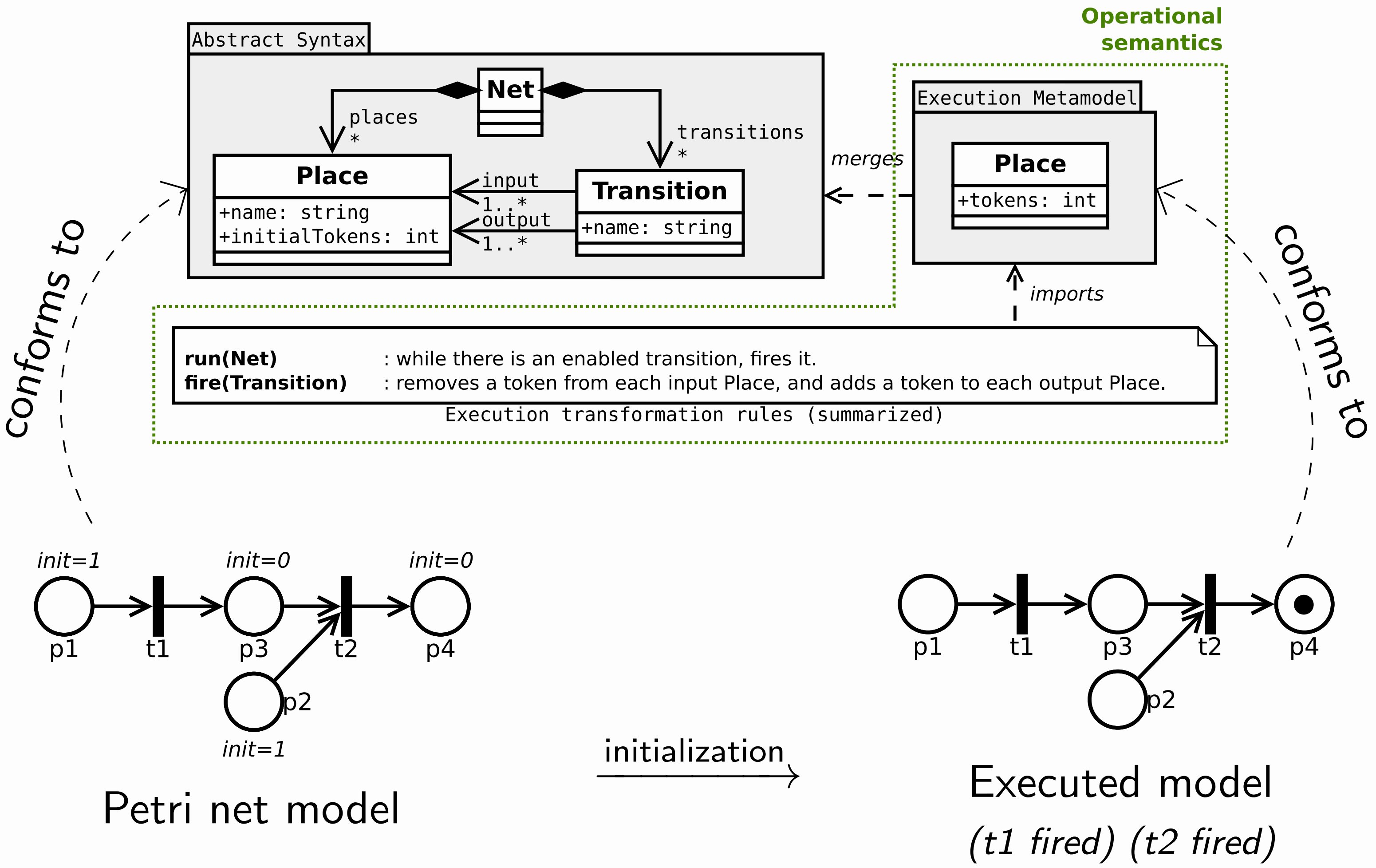
Example of (executable) DSL



Example of (executable) DSL

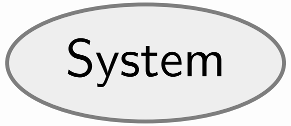


Example of (executable) DSL

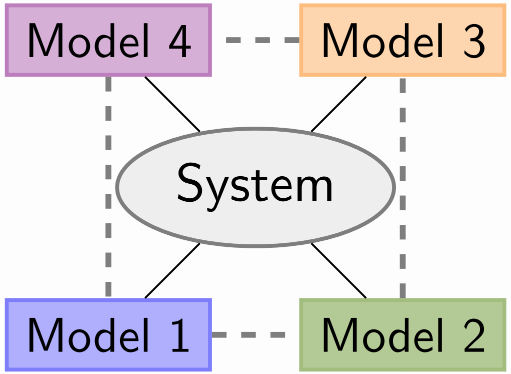


Generic tooling: why and how

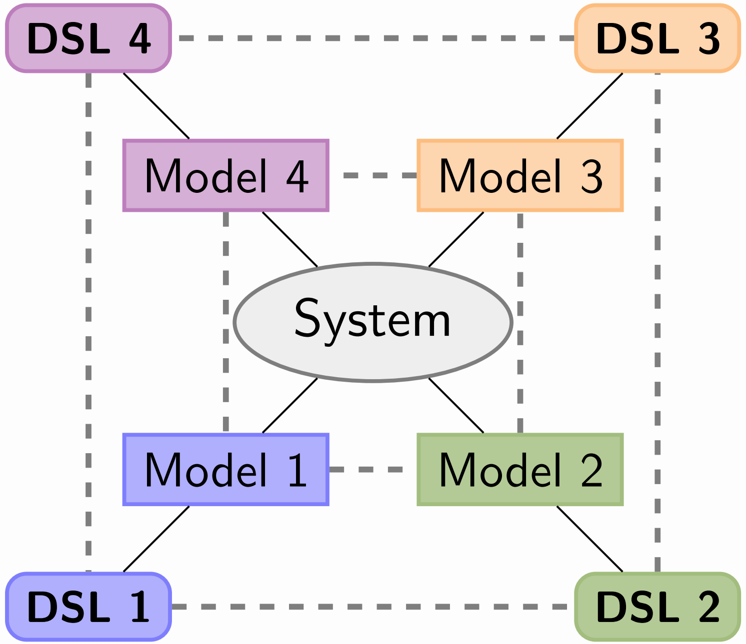
Let's start over...



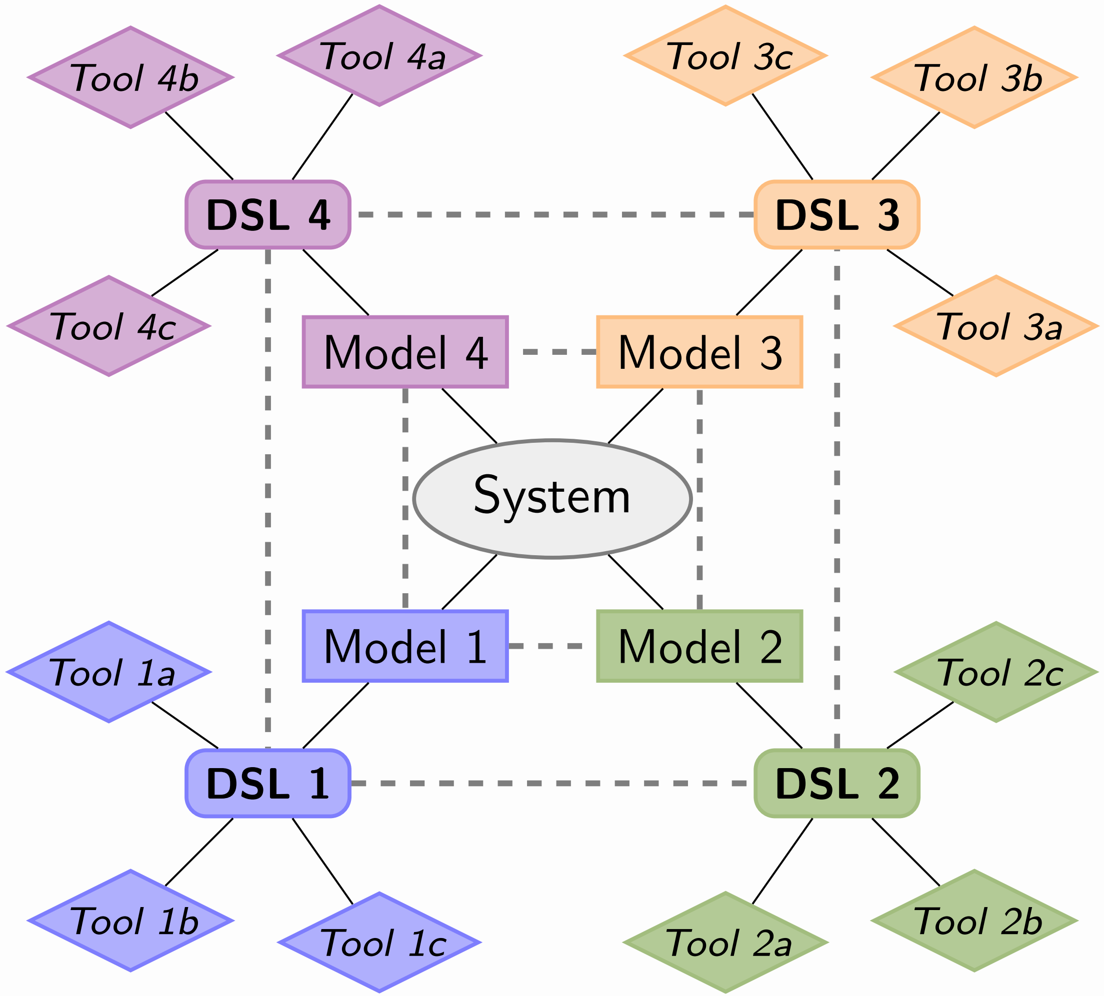
Let's start over...



Let's start over...

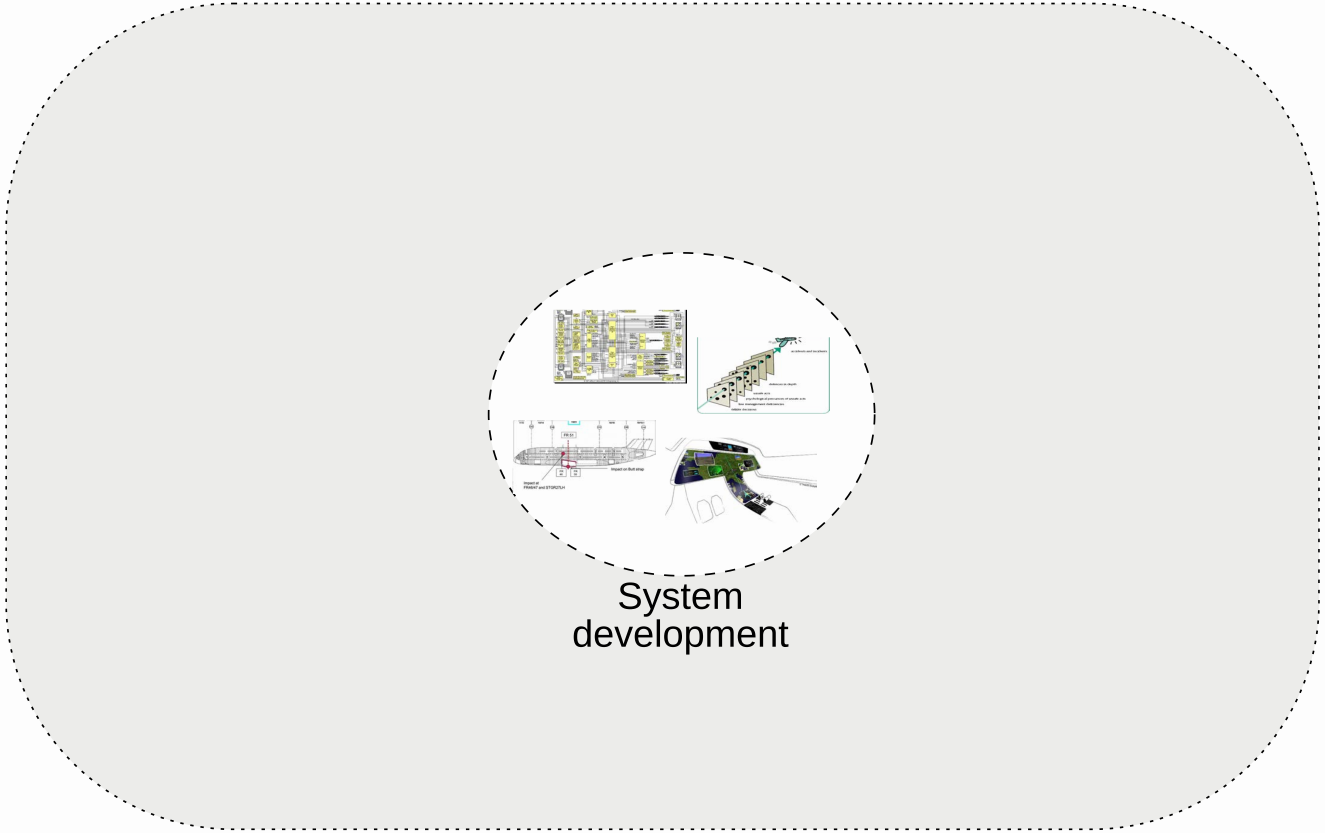


Let's start over...

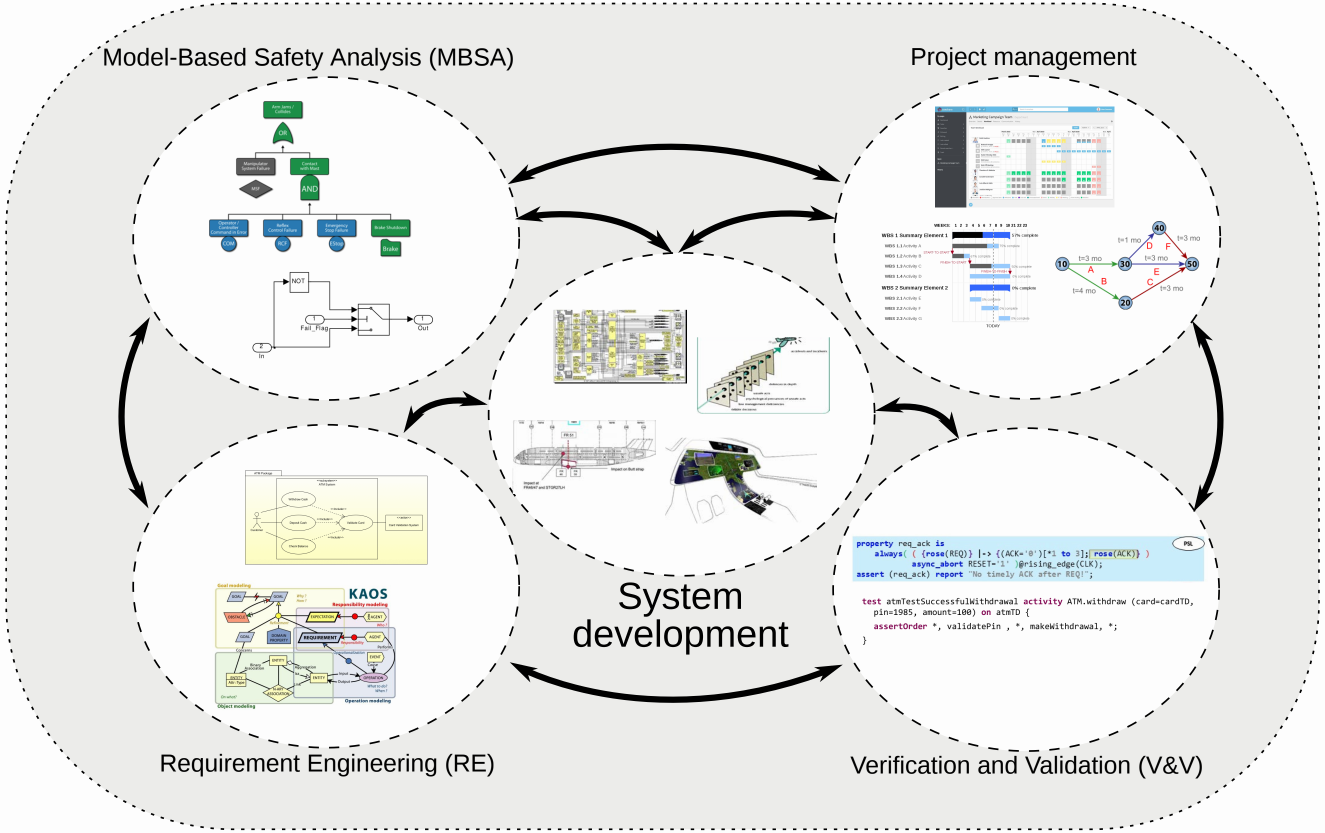


And now a step back...

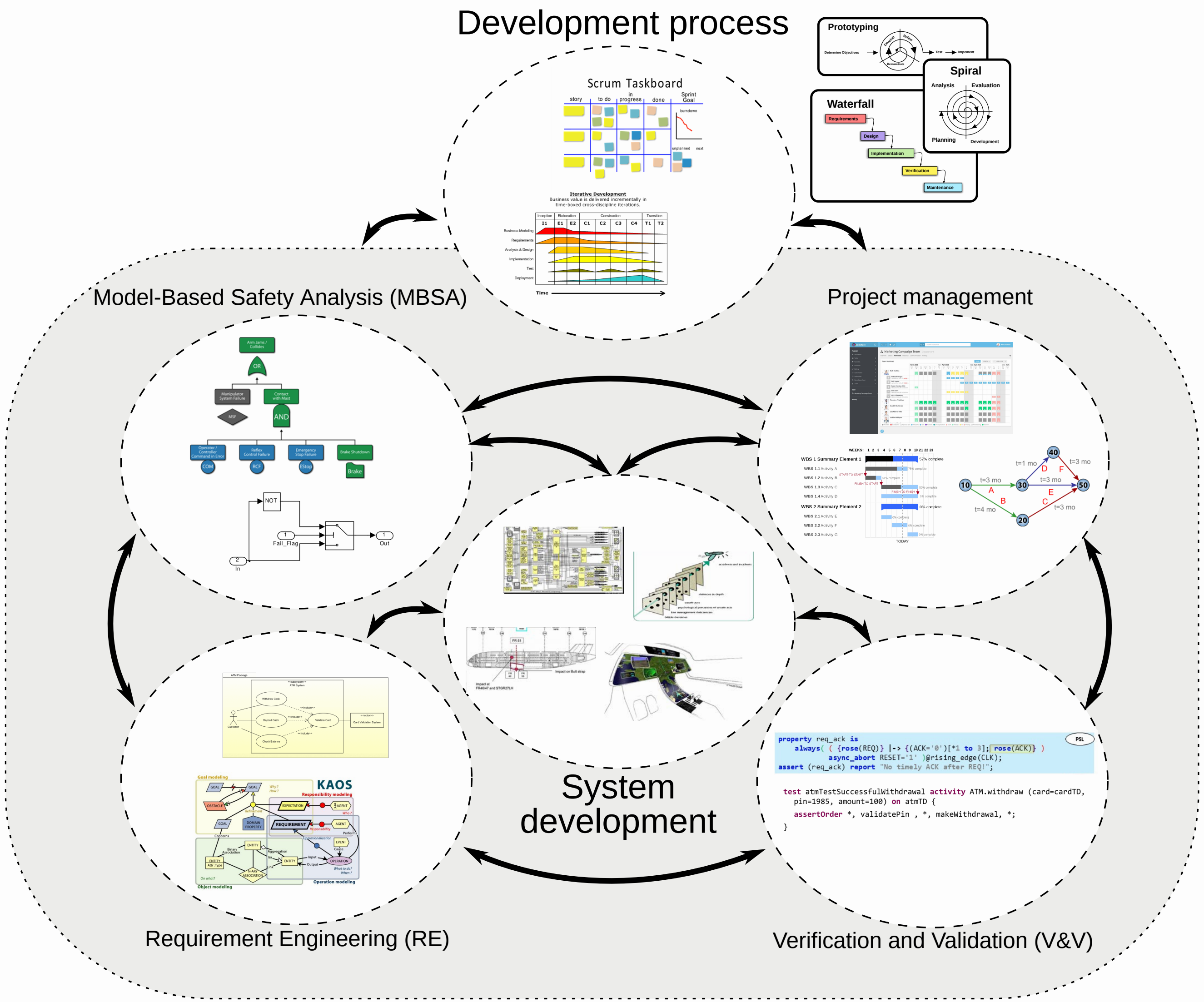
And now a step back...



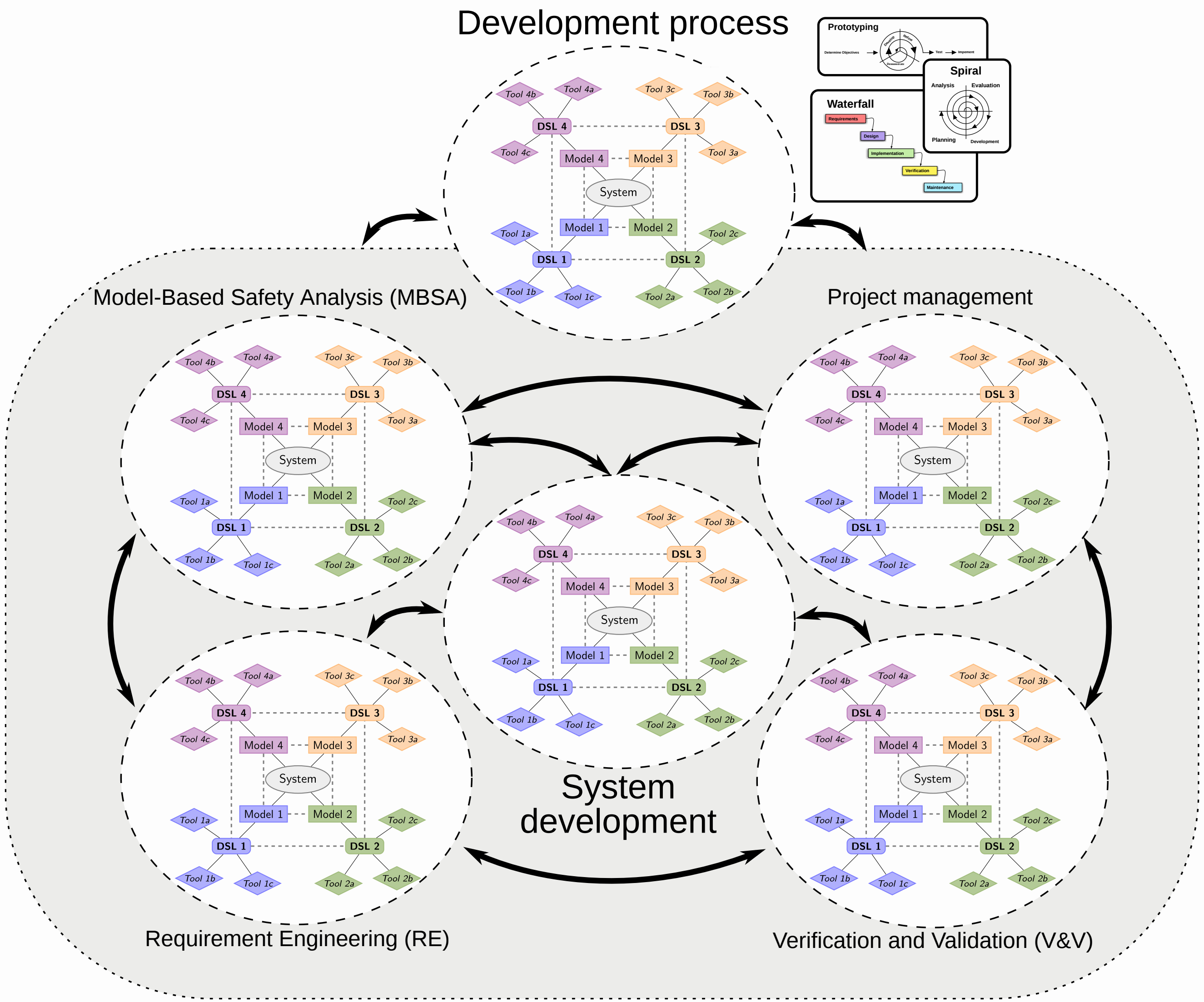
And now a step back...



And now a step back...



And now a step back...



The tool explosion problem

- Developing all aspects of a system requires multiple DSLs...
- Each DSL requires multiple tools (editor, debugger, static analyzer)...
- System development implies multiple contiguous activities, each with its own models, DSLs and tools

Implications

1. Cost: huge amount of tools to develop and maintain
2. Usability: tools must be specialized to activities and DSLs
3. Interoperability: a single tool must cooperate with:
   - Other tools supporting the same DSL
   - Other tools supporting the same activity
   - Other tools supporting contiguous activities

The tool explosion problem

- Developing all aspects of a system requires multiple DSLs...
- Each DSL requires multiple tools (editor, debugger, static analyzer)...
- System development implies multiple contiguous activities, each with its own models, DSLs and tools

Implications

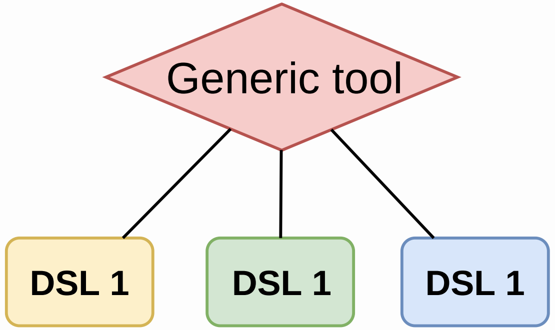
1. Cost: huge amount of tools to develop and maintain
2. Usability: tools must be specialized to activities and DSLs
3. Interoperability: a single tool must cooperate with:
   - Other tools supporting the same DSL
   - Other tools supporting the same activity
   - Other tools supporting contiguous activities

Let's focus on reducing *cost!*

Generic tools to the rescue

Idea

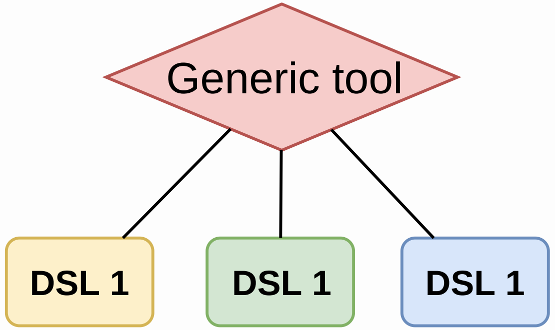
- Instead of restricting a tool to a given DSL, can we make tools compatible with a *wide range of DSL*?
- In other words: can we make truly generic tools?



Generic tools to the rescue

Idea

- Instead of restricting a tool to a given DSL, can we make tools compatible with a *wide range of DSL*?
- In other words: can we make truly generic tools?



Problem

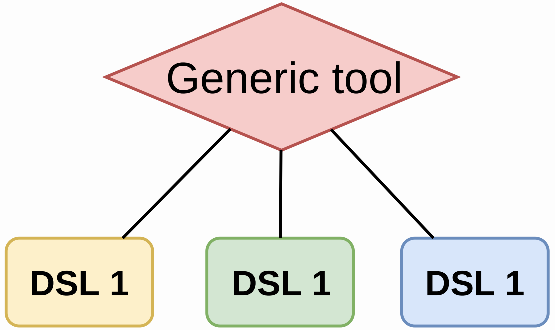
But a generic tool was not coded with domain knowledge, how can it provide relevant services?

- a state machine editor knows that it can draw states and transitions
- ... what does a generic editor know?

Generic tools to the rescue

Idea

- Instead of restricting a tool to a given DSL, can we make tools compatible with a *wide range of DSL*?
- In other words: can we make truly generic tools?



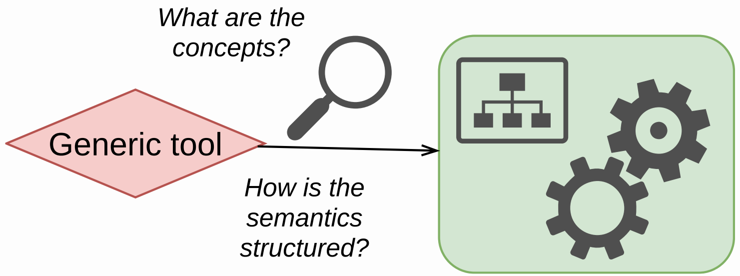
Problem

But a generic tool was not coded with domain knowledge, how can it provide relevant services?

- a state machine editor knows that it can draw states and transitions
- ... what does a generic editor know?

Solution

A generic tool can learn domain knowledge on the fly, ie. a generic tool can "open" the DSL definition and understand its content!



Basic recipe to create a generic tool

Prerequisite: the DSL definition must be analysable (*ie.* whitebox)

1. Define the services of the tool in a language-agnostic fashion
2. Scope which families of DSLs can be targeted by the tool (*eg. "only metamodel-based DSLs"*)
   - this is mandatory for the tool to be able to discover the DSL content automatically
3. Enrich the DSL definition with necessary non-explicit information

- Two main categories of generic tools:
  - *tool generators*, which interpret the DSL definition to generate a DSL-specific tool
  - *regular tools*, which interpret the DSL definition at load-time or at design-time to provide services

A well-known generic tool: Xtext

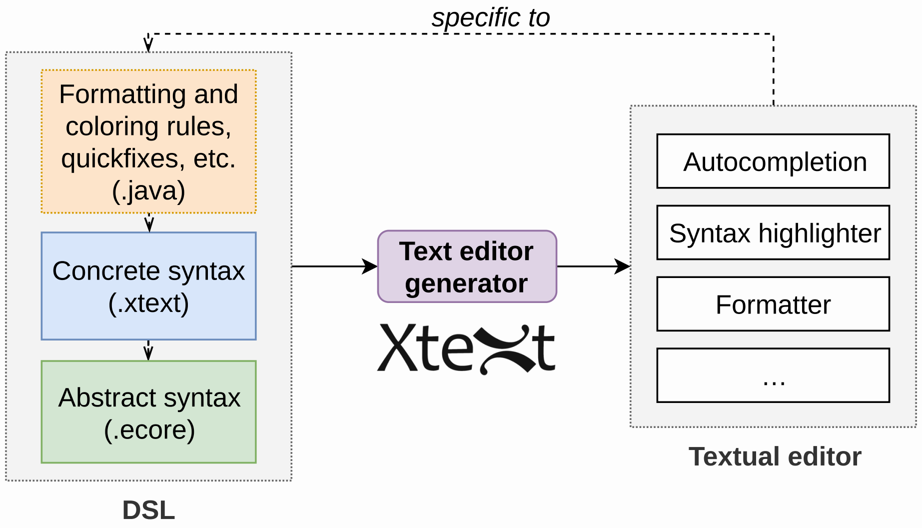
Very popular framework that can be used to:

1. define the textual concrete syntax of DSLs,
2. generate a full-fledged textual editor from the DSL definition.

A well-known generic tool: Xtext

Very popular framework that can be used to:

1. define the textual concrete syntax of DSLs,
2. generate a full-fledged textual editor from the DSL definition.



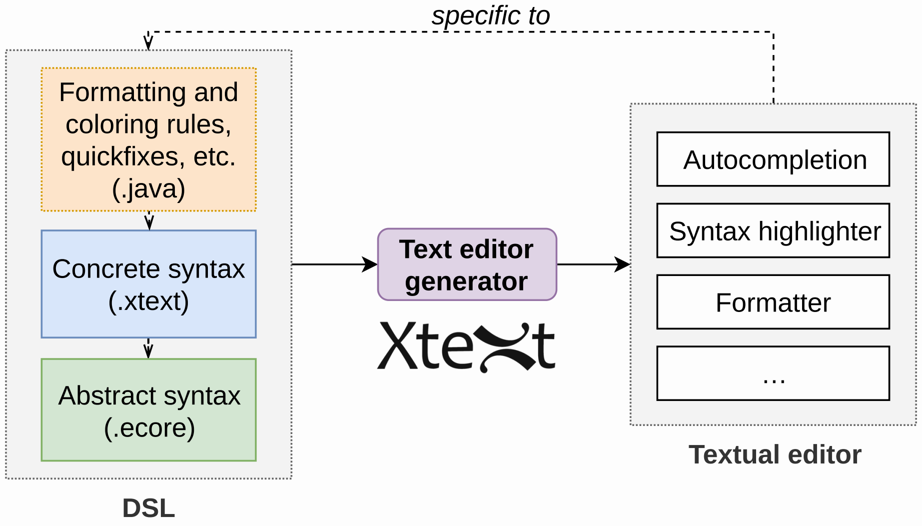
Characteristics

- Generic *tool generator*: the DSL definition is read only once to produce a DSL-specific tool
- *Scope:* DSLs defined using Ecore and Xtext
- *Required enrichment of the DSL:* formatting and coloring rules, quickfix system, builder, etc.

A well-known generic tool: Xtext

Very popular framework that can be used to:

1. define the textual concrete syntax of DSLs,
2. generate a full-fledged textual editor from the DSL definition.



Characteristics

- Generic *tool generator*: the DSL definition is read only once to produce a DSL-specific tool
- *Scope:* DSLs defined using Ecore and Xtext
- *Required enrichment of the DSL:* formatting and coloring rules, quickfix system, builder, etc.

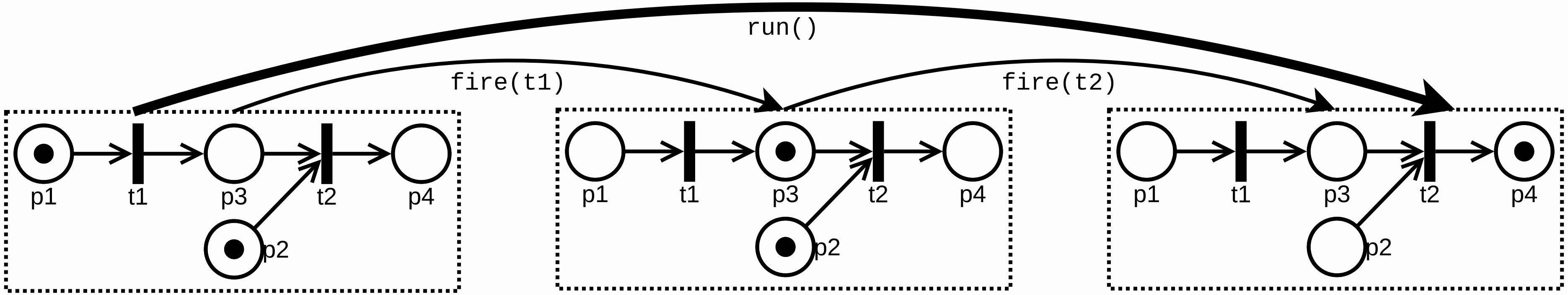
Focus of next parts: *dynamic* generic tools used at execution time

Case 1: Generic trace management

Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, Benoit Baudry. *Advanced and efficient execution trace management for executable domain-specific modeling languages*. SoSym 2017.

Execution traces

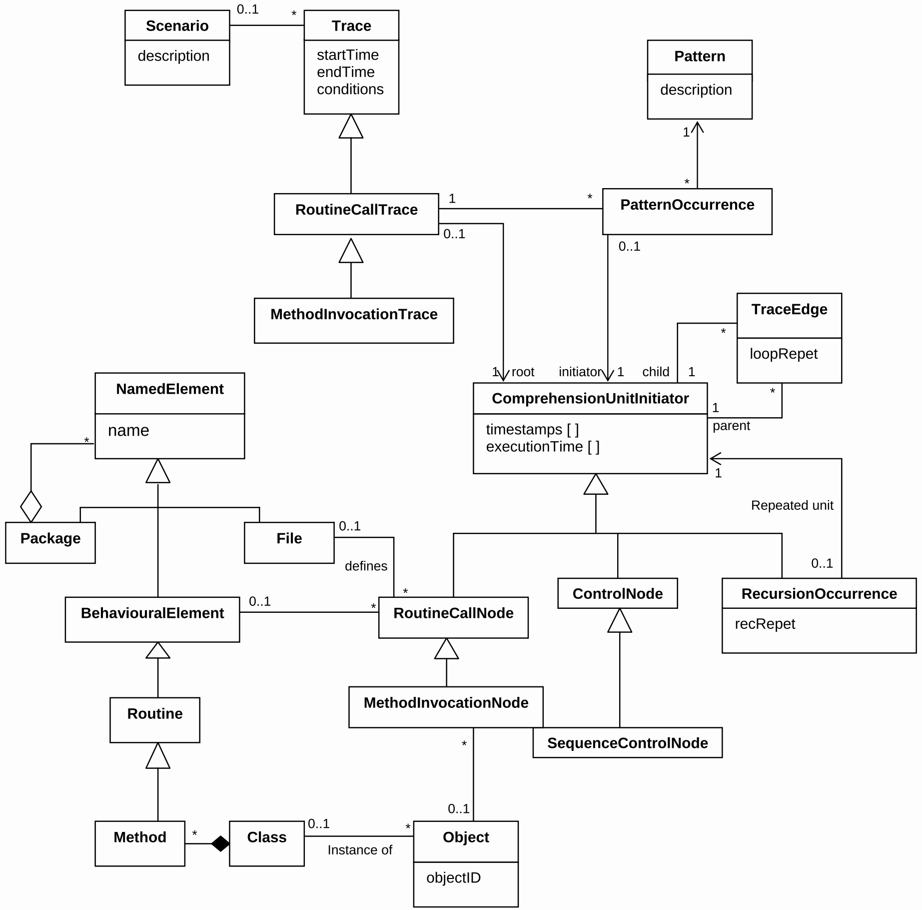
Example of a Petri net execution trace:



Problem

- A wide range of dynamic verification and validation approaches relies on execution traces (runtime monitoring, semantic differencing, model checking, ...)
- How can we represent executions in order to analyze them?

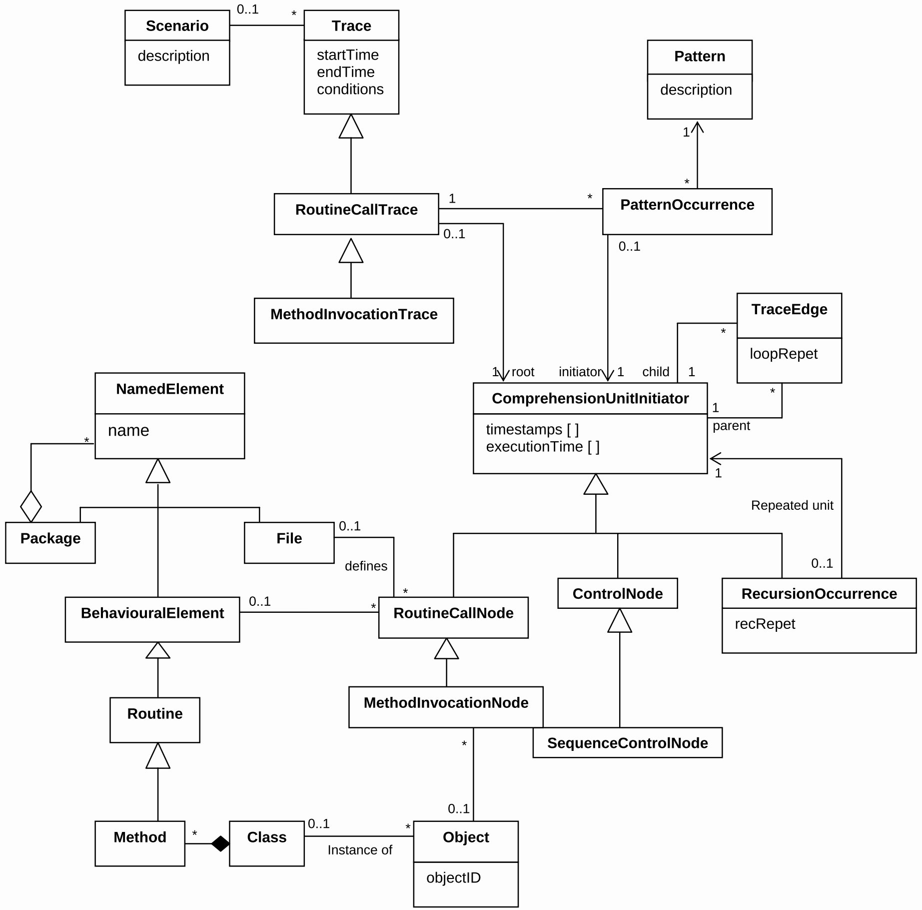
Example of domain-specific tracing



Compact Trace Format (CTF) [Hamou-Lhadj2012]

- Trace format designed for object-oriented programming languages
- Aimed towards lossless compression of traces
- Concepts such as *class*, *routine*, *package*, ...
- Cannot be used to trace other kinds of languages (*eg.* Petri nets)

Example of domain-specific tracing



Compact Trace Format (CTF) [Hamou-Lhadj2012]

- Trace format designed for object-oriented programming languages
- Aimed towards lossless compression of traces
- Concepts such as *class*, *routine*, *package*, ...
- Cannot be used to trace other kinds of languages (*eg.* Petri nets)

We must re-think tracing in a *language-agnostic* fashion

Towards *generic* execution trace management

- Tracing in a specific context relies on specific concepts:
  - A Java trace is composed of *method calls* and *heaps snapshots*
  - An activity diagram trace is a sequence of *activated nodes*

Towards *generic* execution trace management

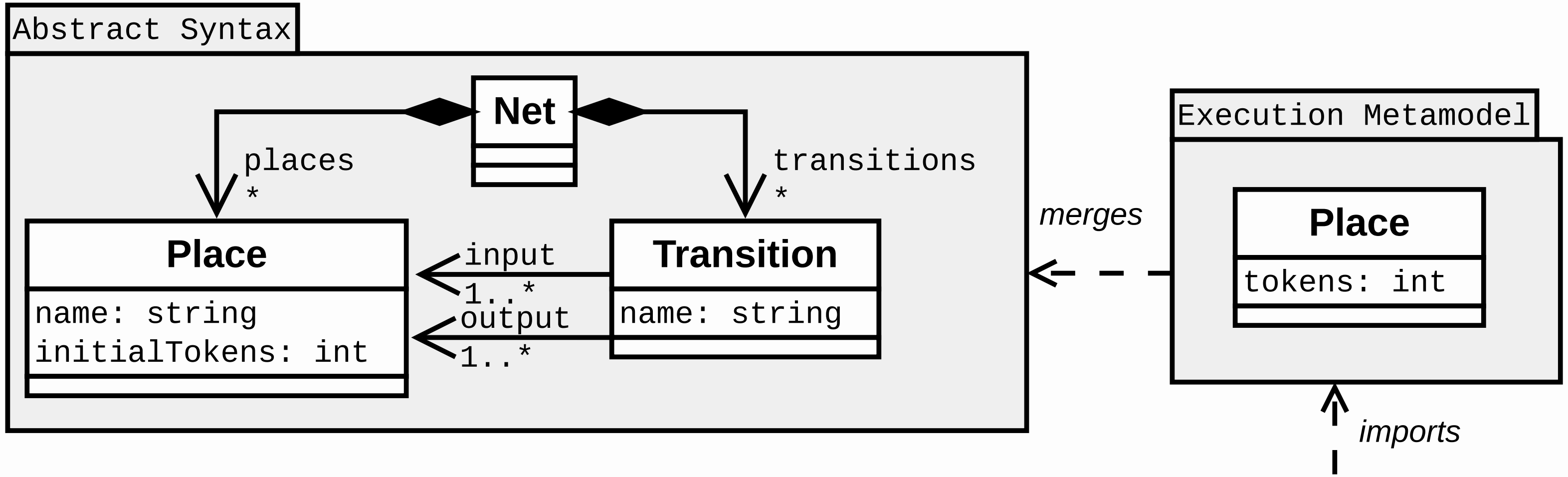
- Tracing in a specific context relies on specific concepts:
  - A Java trace is composed of *method calls* and *heaps snapshots*
  - An activity diagram trace is a sequence of *activated nodes*
- Tracing for any DSL (ie. any context) requires generic concepts, such as:
  - An execution state stores the values of the dynamic parts of the model (e.g. tokens)
  - An execution step is the application of a execution rule of the semantics (eg. fire)

Towards *generic* execution trace management

- Tracing in a specific context relies on specific concepts:
  - A Java trace is composed of *method calls* and *heaps snapshots*
  - An activity diagram trace is a sequence of *activated nodes*
- Tracing for any DSL (ie. any context) requires generic concepts, such as:
  - An execution state stores the values of the dynamic parts of the model (e.g. tokens)
  - An execution step is the application of a execution rule of the semantics (eg. fire)
- Consequently: we need more information in the DSL definition
  - what are the dynamic parts (*ie.* the execution state definition)?
  - what are the possible execution steps?

1 – Enrichment of the DSL with steps and states

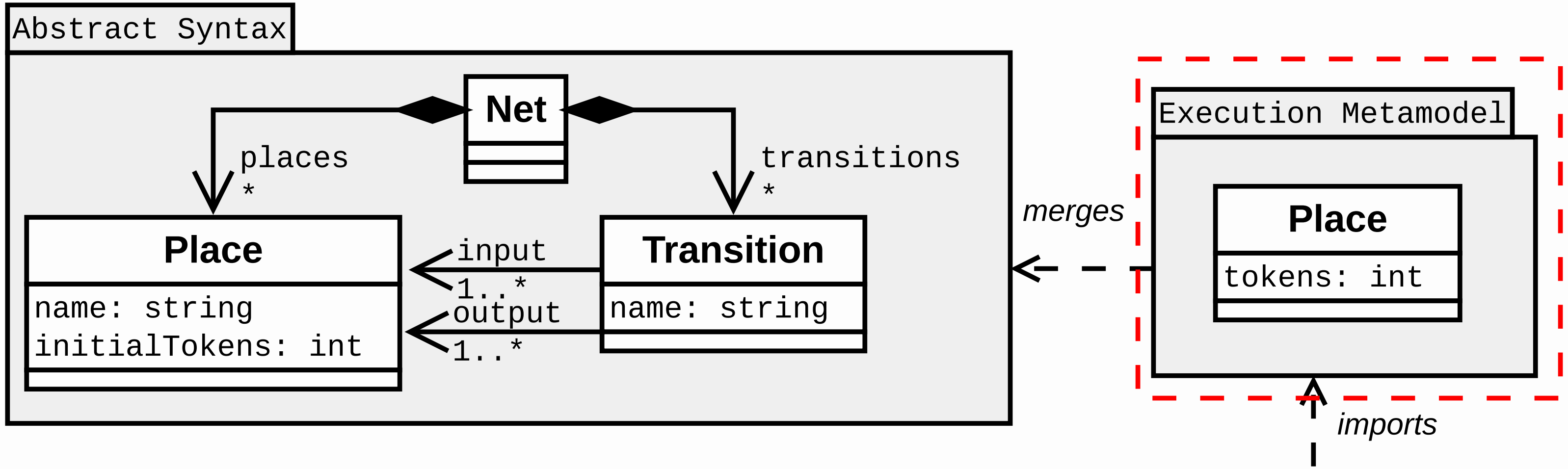
Enrichment



run(Net)	: while there is an enabled transition, fires it.
isEnabled(Transition)	: returns true if tokens > 0 for each input Place, false otherwise.
fire(Transition)	: removes a token from each input Place and adds one to each output Place.
addToken(Place)	: adds a token to a Place
removeToken(Place)	: removes a token from a Place

Execution transformation rules (summarized)

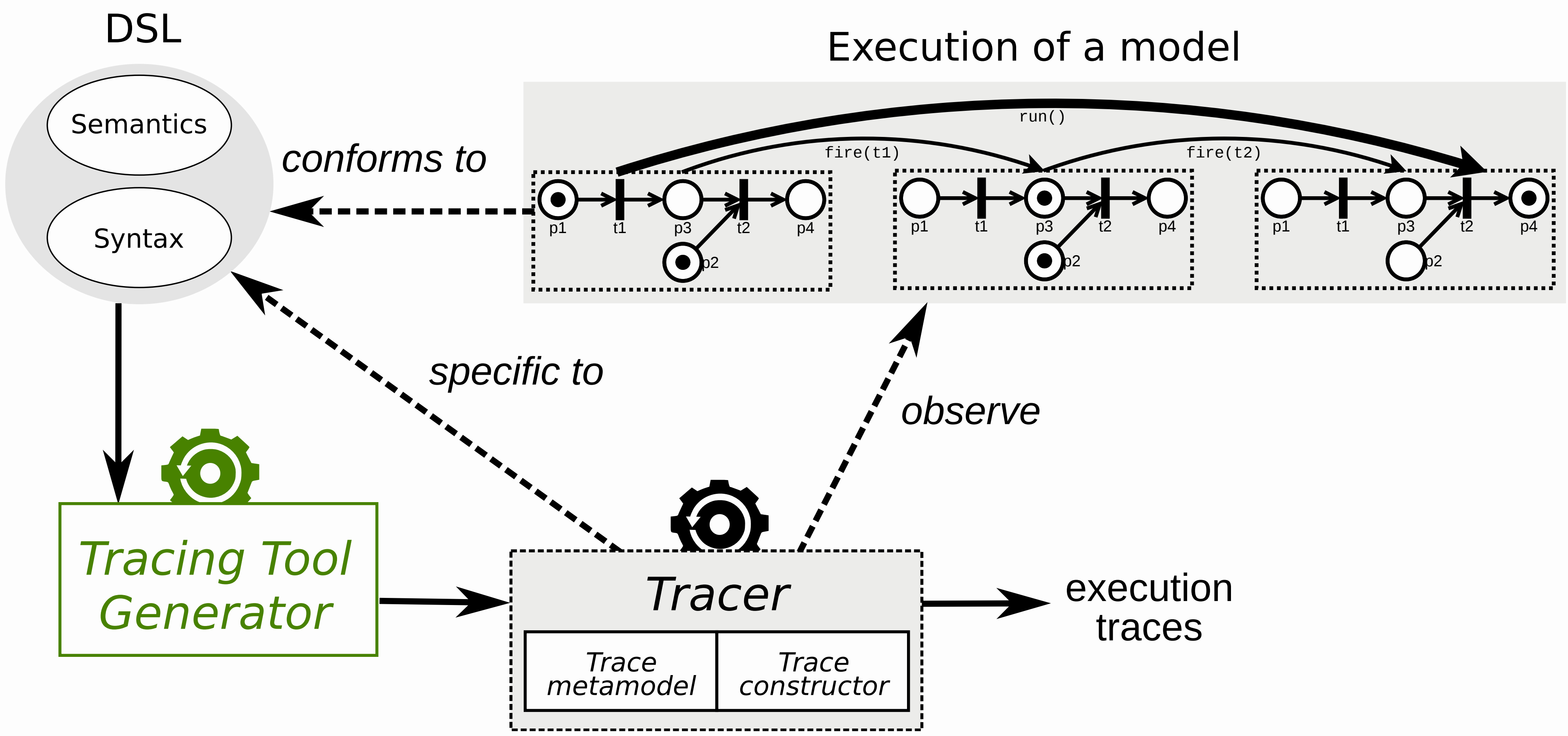
Enrichment



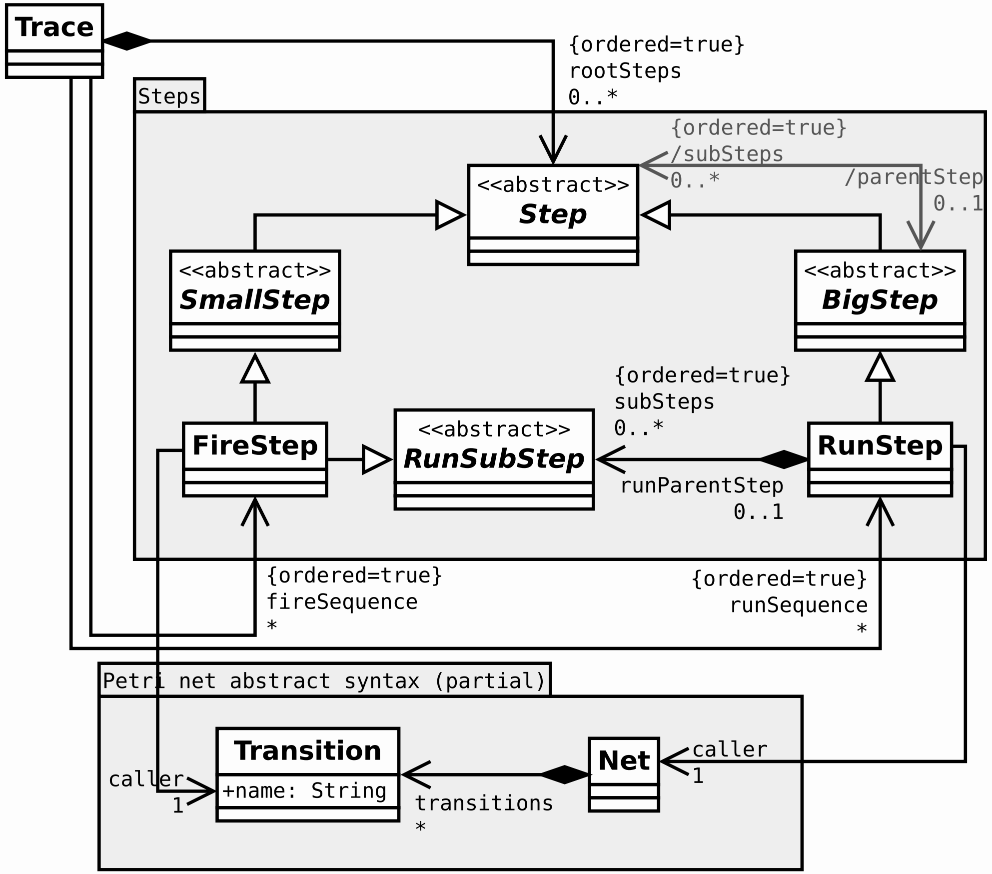
@Step	**run(Net)**	: while there is an enabled transition, fires it.
	isEnabled(Transition)	: returns true if tokens > 0 for each input Place, false otherwise.
@Step	**fire(Transition)**	: removes a token from each input Place and adds one to each output Place.
	addToken(Place)	: adds a token to a Place
	removeToken(Place)	: removes a token from a Place

Execution transformation rules (summarized)

2 – Generic generation of a DSL-specific tracer



Trace metamodel generation – Steps concepts

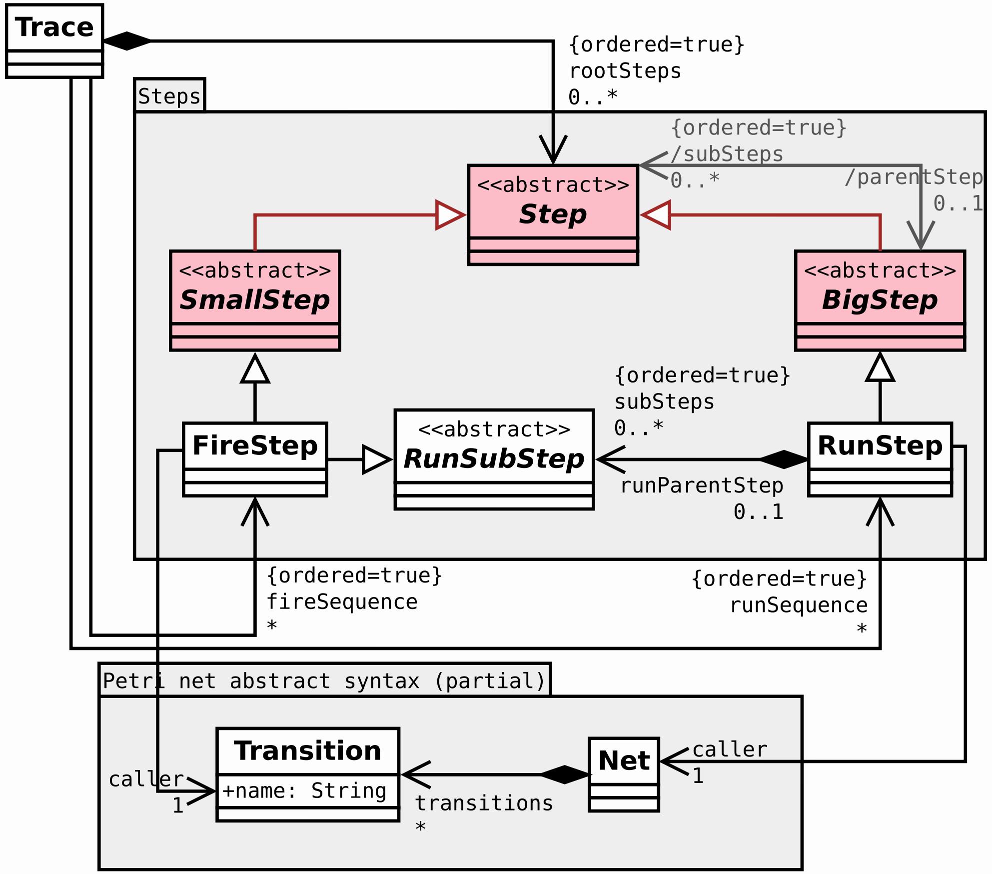


1 Base classes
- **small step** = standalone transformation rule
- **big step** = rule relying on other rules

2 Reification of rules into step classes

3 Steps made accessible as sequences or as a containment tree

Trace metamodel generation – Steps concepts



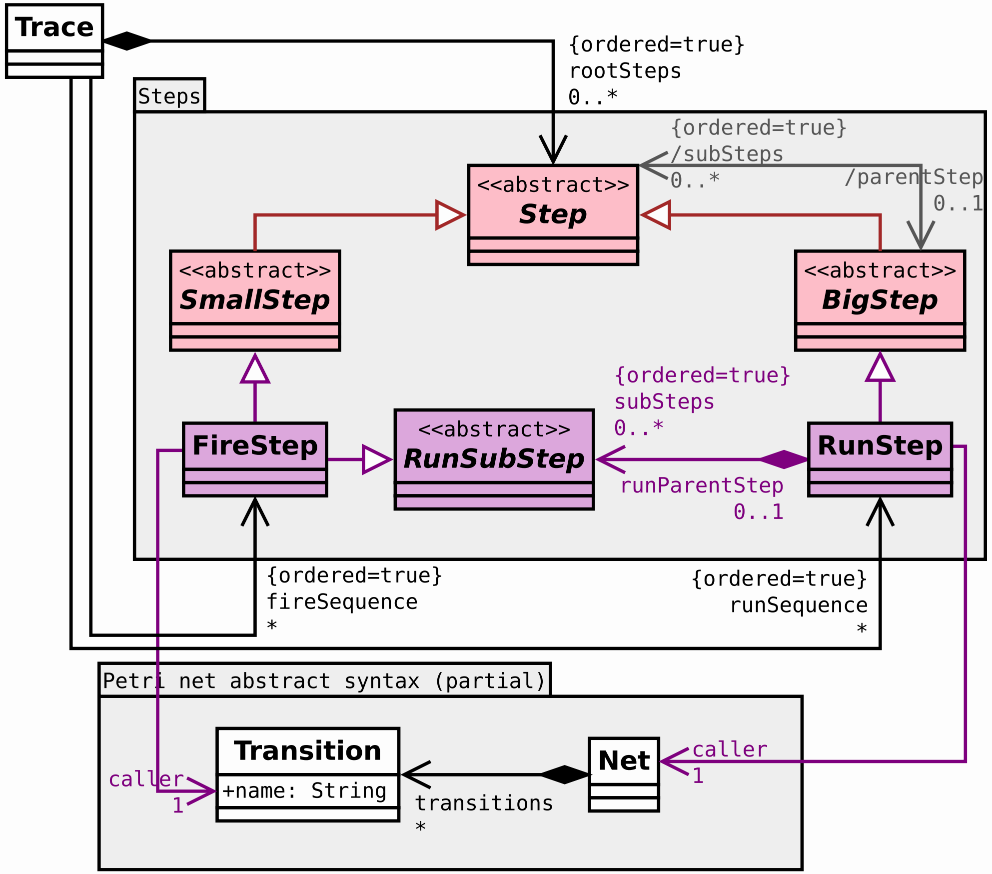
1 Base classes

- **small step** = standalone transformation rule
- **big step** = rule relying on other rules

2 Reification of rules into step classes

3 Steps made accessible as sequences or as a containment tree

Trace metamodel generation – Steps concepts

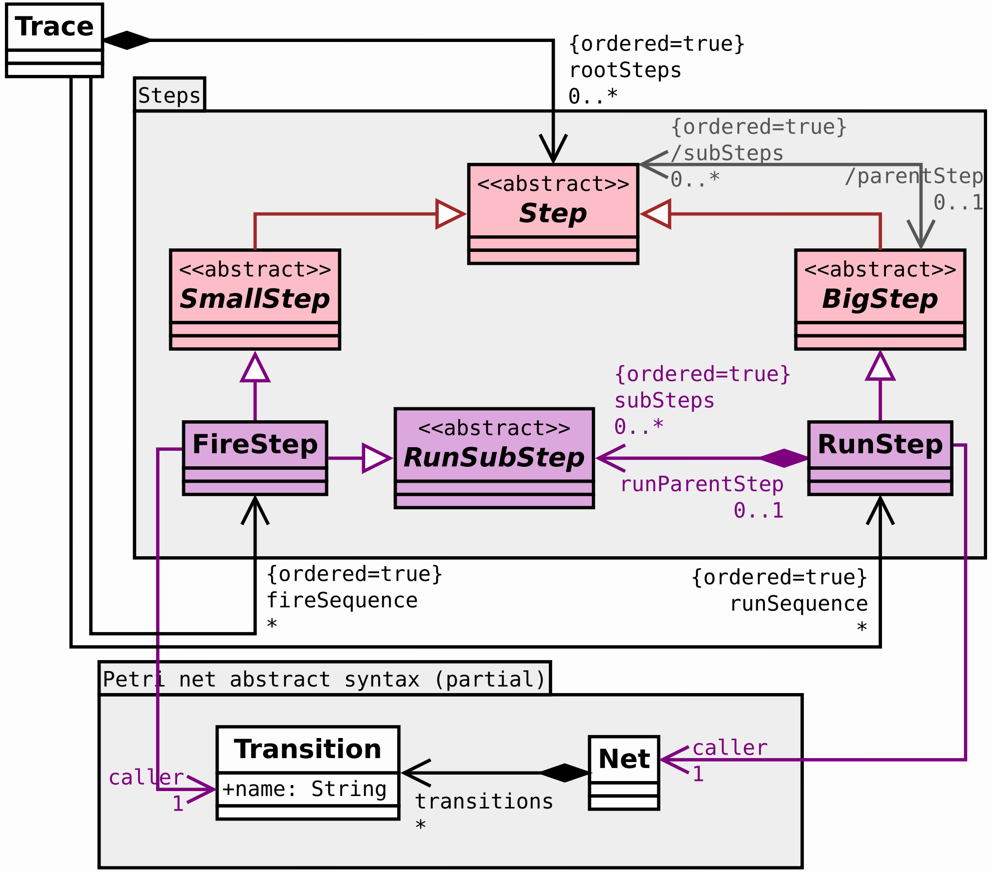


1 Base classes
- **small step** = standalone transformation rule
- **big step** = rule relying on other rules

2 Reification of rules into step classes

3 Steps made accessible as sequences or as a containment tree

Trace metamodel generation – Steps concepts



Step classes are inferred by **static analysis** of the model transformation.

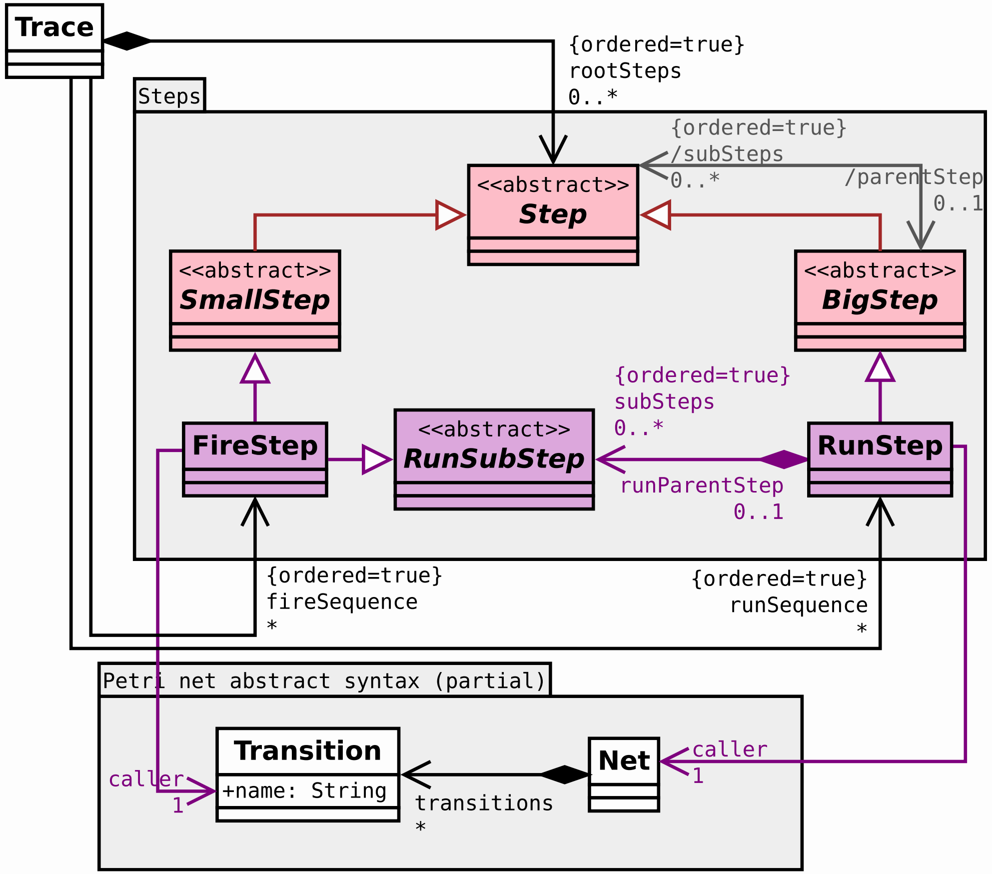
1. Analysis of the code

```
def void fire() {
    ...
}

def void run() {
    while (_self.getNext() != null) {
        _self.getNext().fire()
    }
}
```

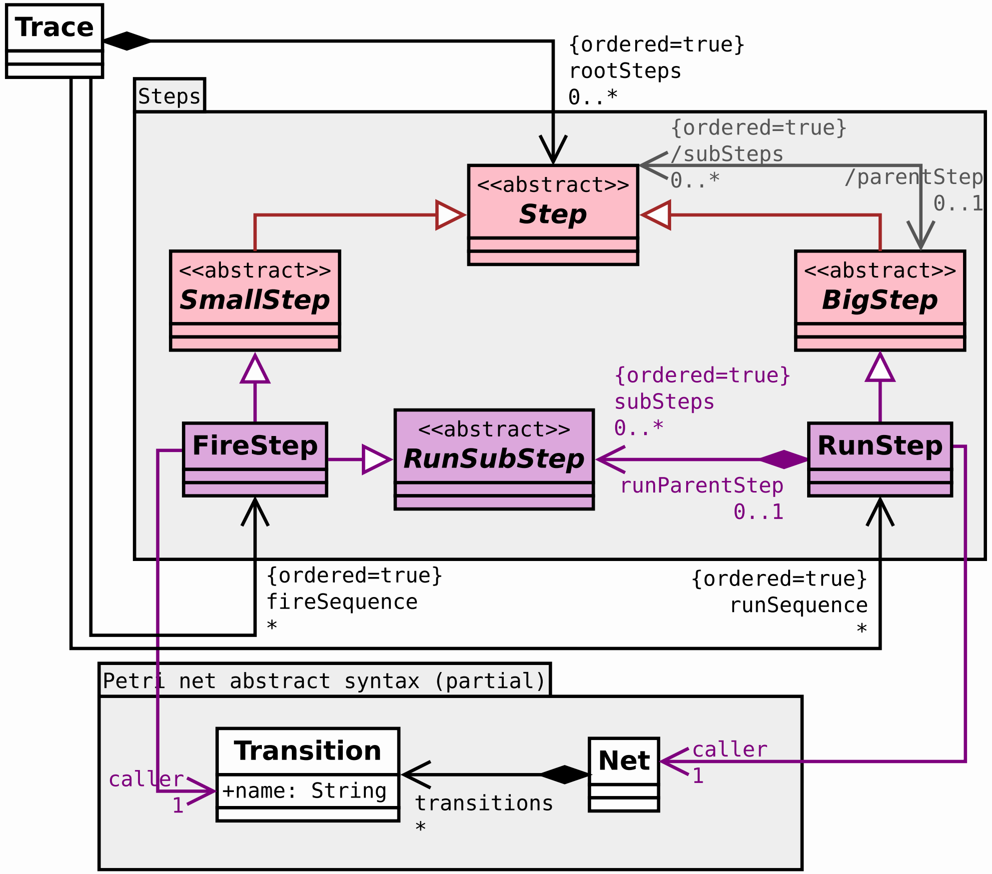
2. Creation of a **call graph**
3. **Pre-processing** of the call graph (e.g. methods overriding)
4. **Discovery** of big/small steps

Trace metamodel generation – Steps concepts



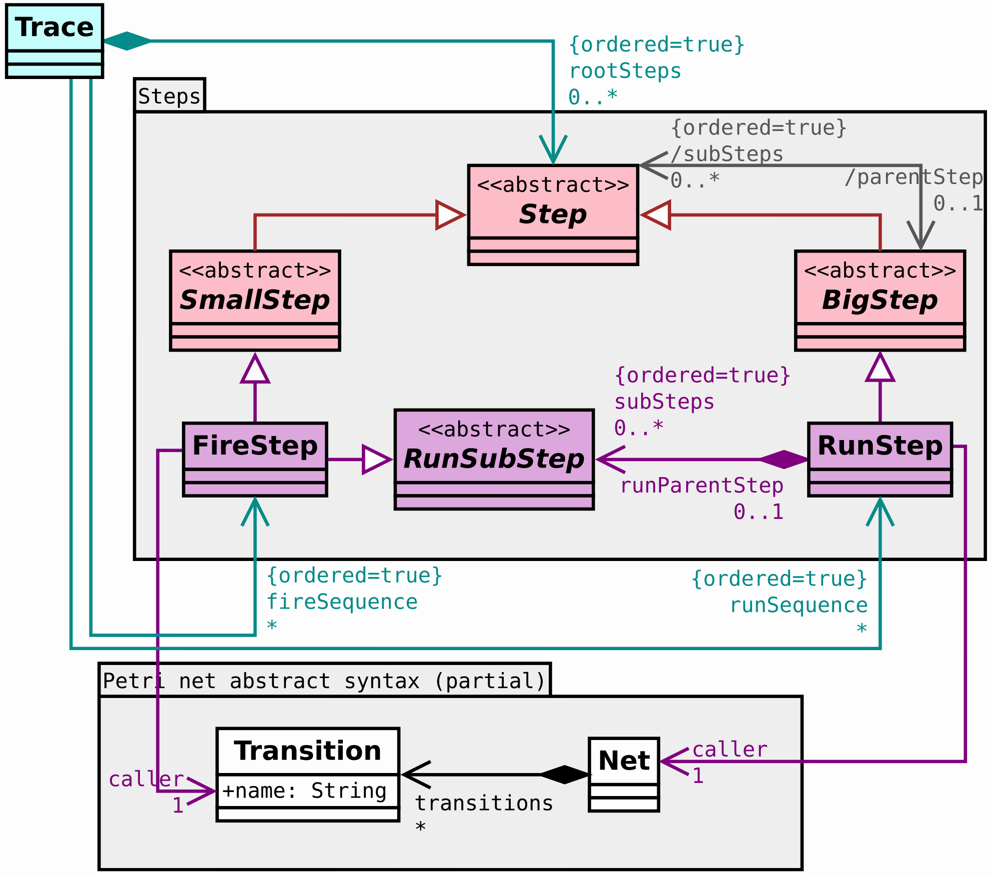
1 Base classes
   - **small step** = standalone transformation rule
   - **big step** = rule relying on other rules
2 Reification of rules into step classes
3 Steps made accessible as sequences or as a containment tree

Trace metamodel generation – Steps concepts



1. Base classes
   - **small step** = standalone transformation rule
   - **big step** = rule relying on other rules
2. Reification of rules into step classes
3. Steps made accessible as sequences or as a containment tree

Trace metamodel generation – Steps concepts

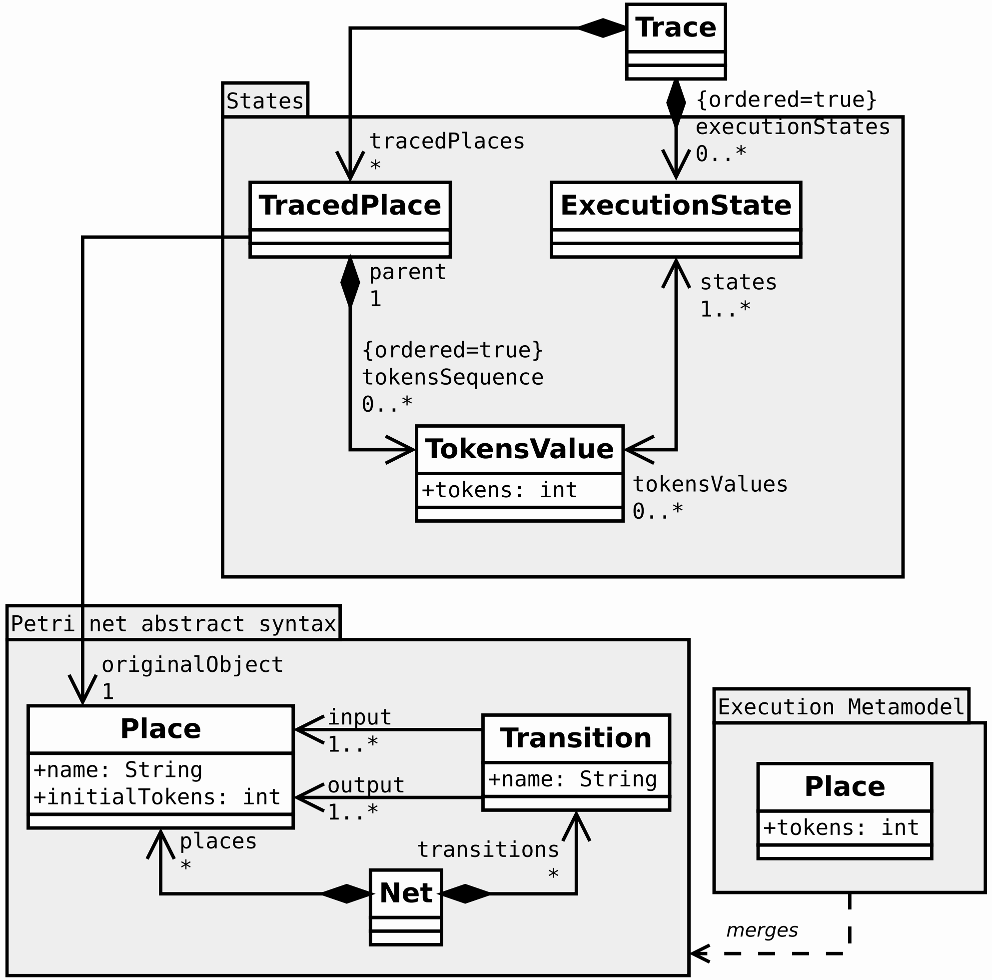


1 Base classes
- **small step** = standalone transformation rule
- **big step** = rule relying on other rules

2 Reification of rules into step classes

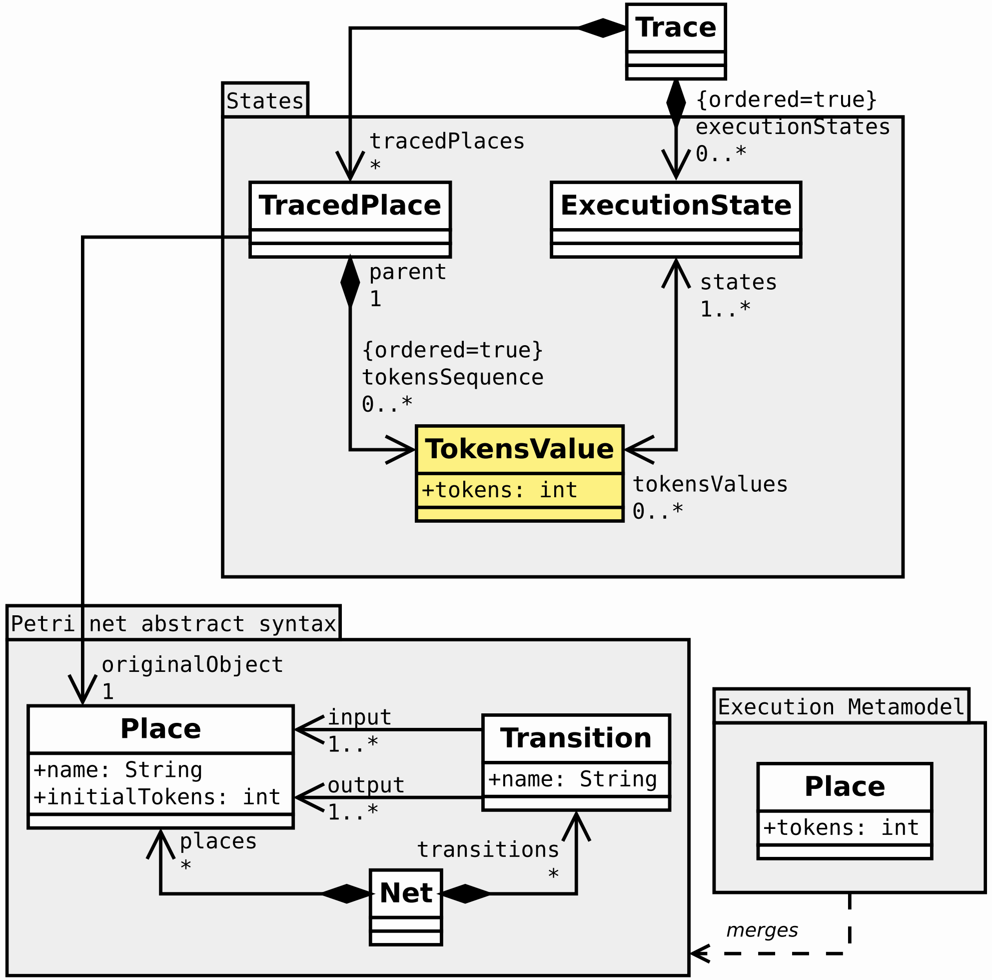
3 Steps made accessible as sequences or as a containment tree

Trace metamodel generation – States concepts



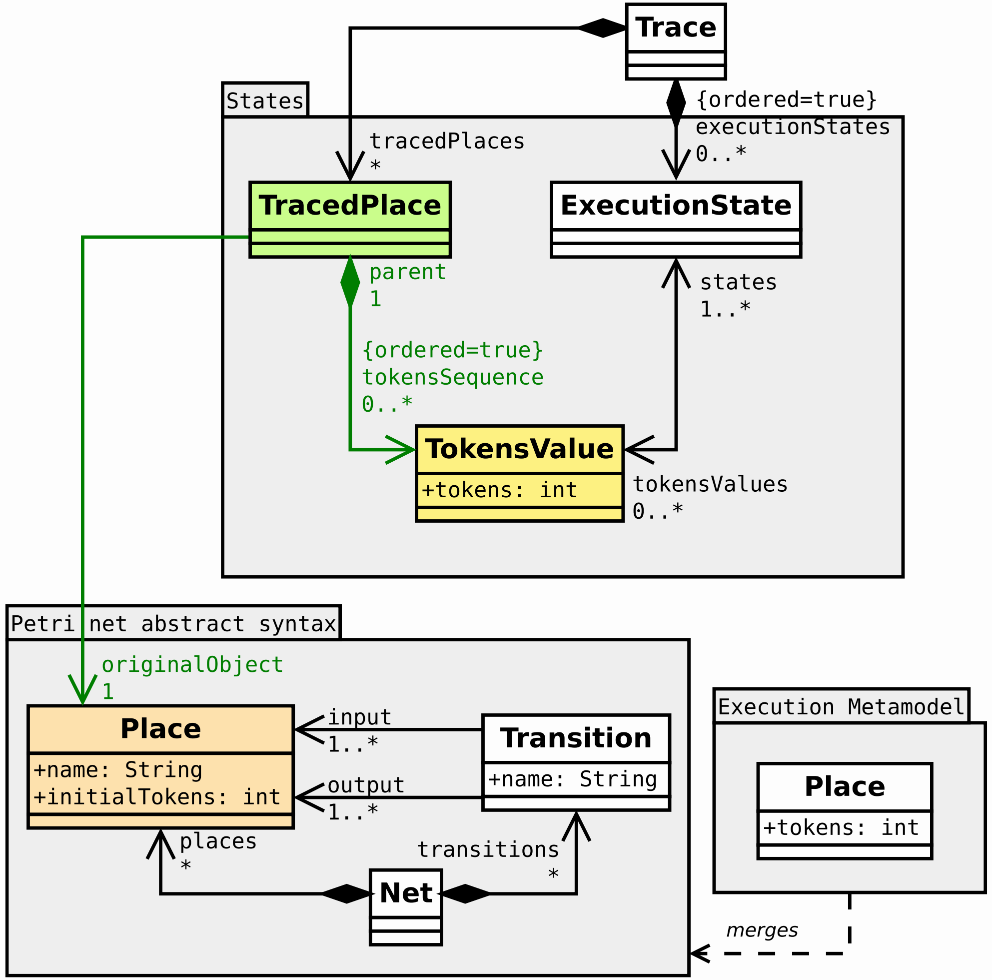
1. Reification of mutable properies into value classes
2. Values stored as sequences , for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

Trace metamodel generation – States concepts



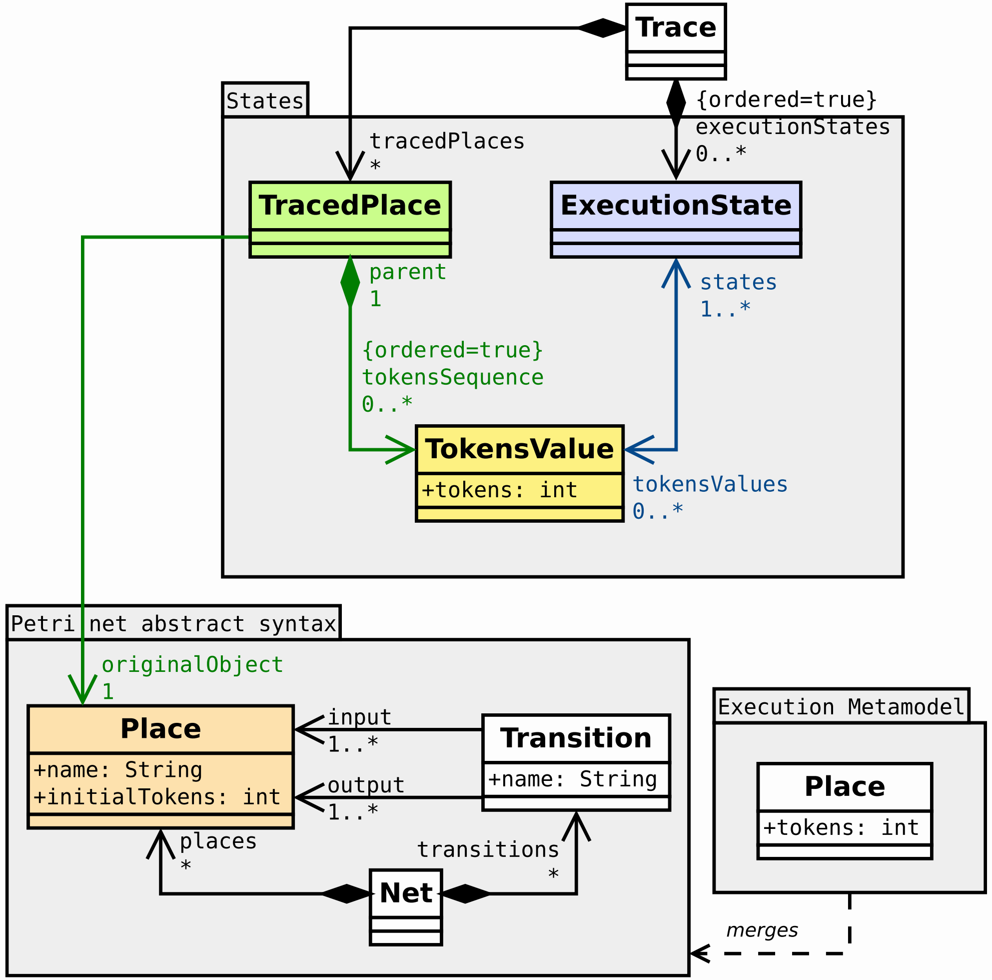
1. Reification of mutable properies into value classes
2. Values stored as sequences , for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

Trace metamodel generation – States concepts



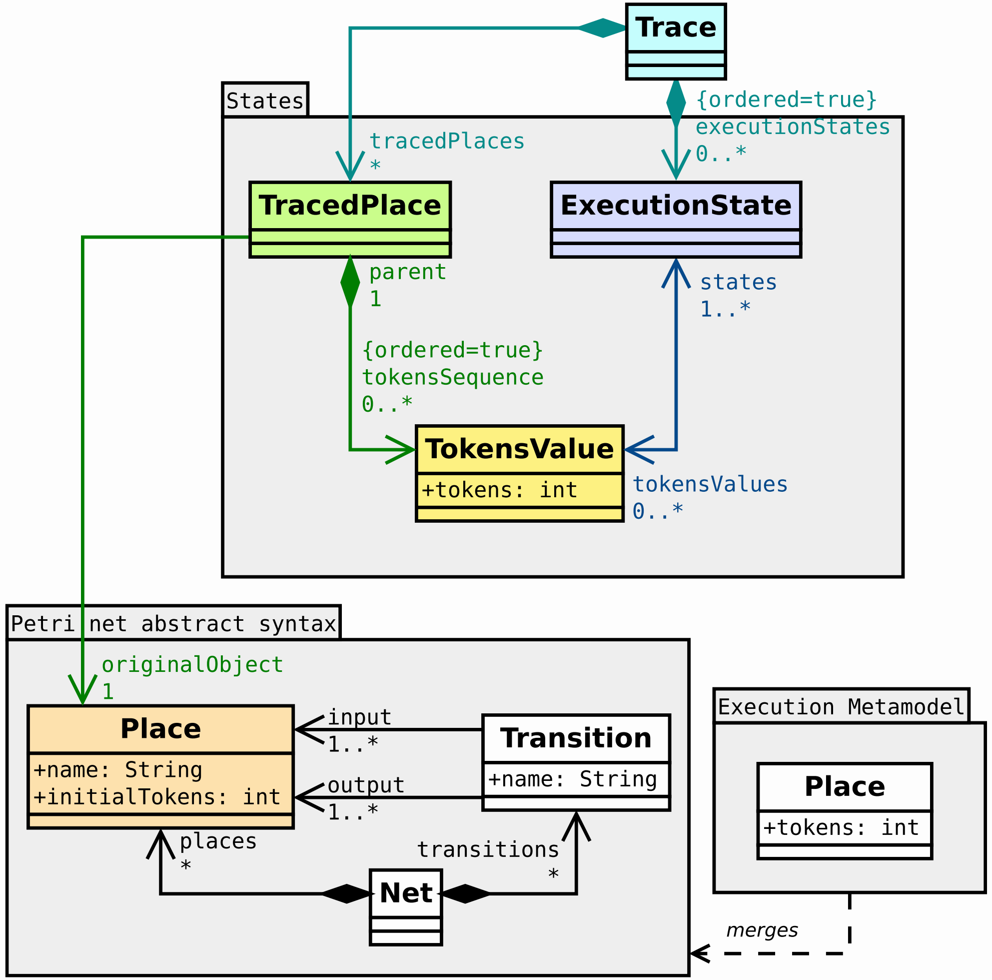
1. Reification of mutable properies into value classes
2. Values stored as sequences, for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

Trace metamodel generation – States concepts



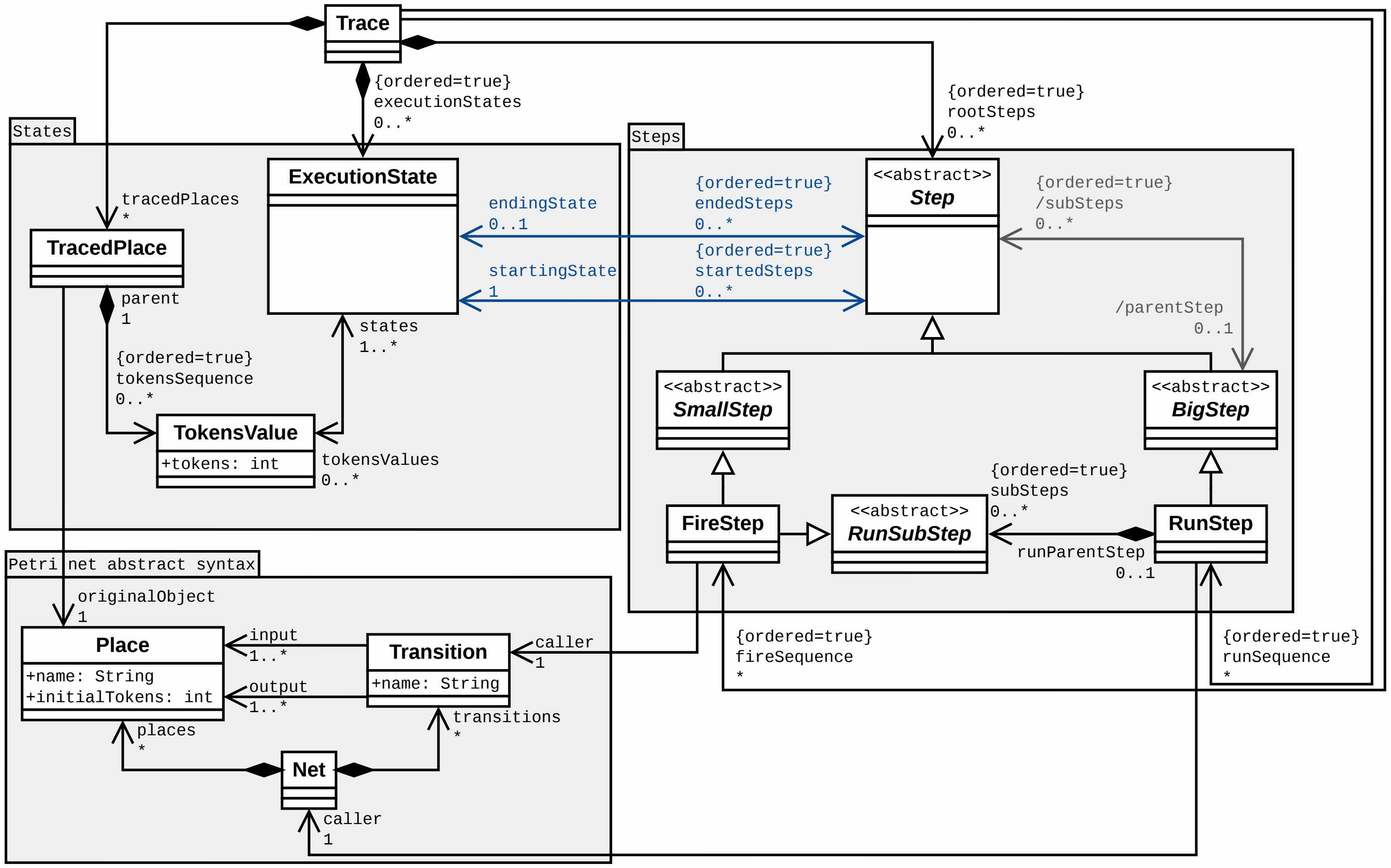
1. Reification of mutable properies into value classes
2. Values stored as sequences, for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

Trace metamodel generation – States concepts



1. Reification of mutable properies into value classes
2. Values stored as sequences, for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

Resulting Petri nets trace metamodel



Resulting Petri nets trace constructor (excerpt)

Algorithm 6: *addState* generated for Petri nets

Input:

$root$	:	root Trace object
$model_{exe}$	:	the model being executed
map_{traced}	:	map with the traced object of each object of the executed model

begin
  $changes \leftarrow \text{getFieldChanges}()$
  $state_{current} \leftarrow root.\texttt{executionStates}.last()$
  if $state_{current} = null$ **then**
    $\text{addInitialState}(root, model_{exe}, map_{traced})$
  else if $changes \neq \varnothing$ **then**
    $state_{new} \leftarrow \text{copyState}(state_{current})$
    $root.\texttt{executionStates}.\text{add}(newState)$
    foreach $fieldChange \in changes$ **do**
      $o \leftarrow fieldChange.\texttt{changedObject}$
      $traced_o \leftarrow map_{traced}(o)$
      if $o.\text{is}(\mathsf{Place})$ **then**
        $p \leftarrow fieldChange.\texttt{changedProperty}$
        if $p.\text{is}(\mathsf{Place}.\texttt{tokens})$ **then**
          $v_{new} \leftarrow \text{createObject}(\mathsf{TokensValue})$
          $v_{new}.\texttt{tokens} = o.\texttt{tokens}$
          $v_{old} \leftarrow traced_o.\texttt{tokensSequence}.\text{last}()$
          $state_{new}.\texttt{tokensValues}.\text{remove}(v_{old})$
          $state_{new}.\texttt{tokensValues}.\text{add}(v_{new})$
          $traced_o.\texttt{tokensSequence}.\text{add}(v_{new})$

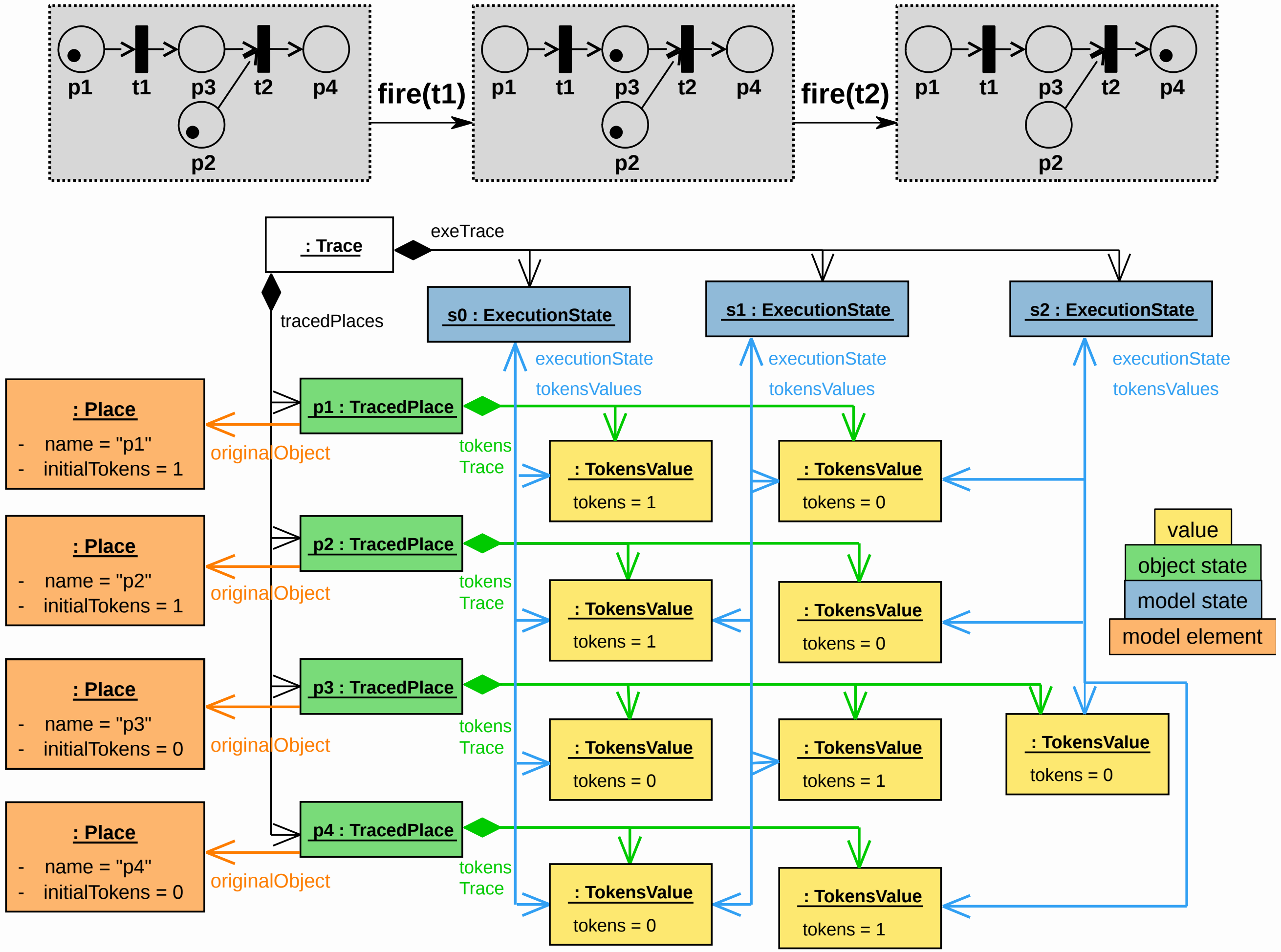
Algorithm 7: *addStep* generated for Petri nets

Input:

$root$	:	root Trace object
$stepRuleID$	:	ID of the step rule
$stepRuleParams$	:	parameters given to the rule
$stack_{steps}$	:	stack of all ongoing current steps

begin
  $step_{new} \leftarrow null$
  $state_{current} \leftarrow root.\texttt{executionStates}.\text{last}()$
  $step_{current} \leftarrow stack_{steps}.\text{peek}()$
  if $stepRuleID = \text{getRuleID}(\mathsf{RunStep})$ **then**
    $step_{new} \leftarrow \text{createObject}(\mathsf{RunStep})$
    $step_{new}.\texttt{caller} \leftarrow stepRuleParams[O]$
    $root.\texttt{runSequence}.\text{add}(step_{new})$
    $root.\texttt{rootSteps}.\text{add}(step_{new})$
  else if $stepRuleID = \text{getRuleID}(\mathsf{FireStep})$ **then**
    $step_{new} \leftarrow \text{createObject}(\mathsf{FireStep})$
    $step_{new}.\texttt{caller} \leftarrow stepRuleParams[O]$
    $root.\texttt{fireSequence}.\text{add}(step_{new})$
    $step_{current}.\texttt{subSteps}.\text{add}(step_{new})$
  $step_{new}.\texttt{startingState} \leftarrow state_{current}$
  $stack_{steps}.\text{push}(step_{new})$

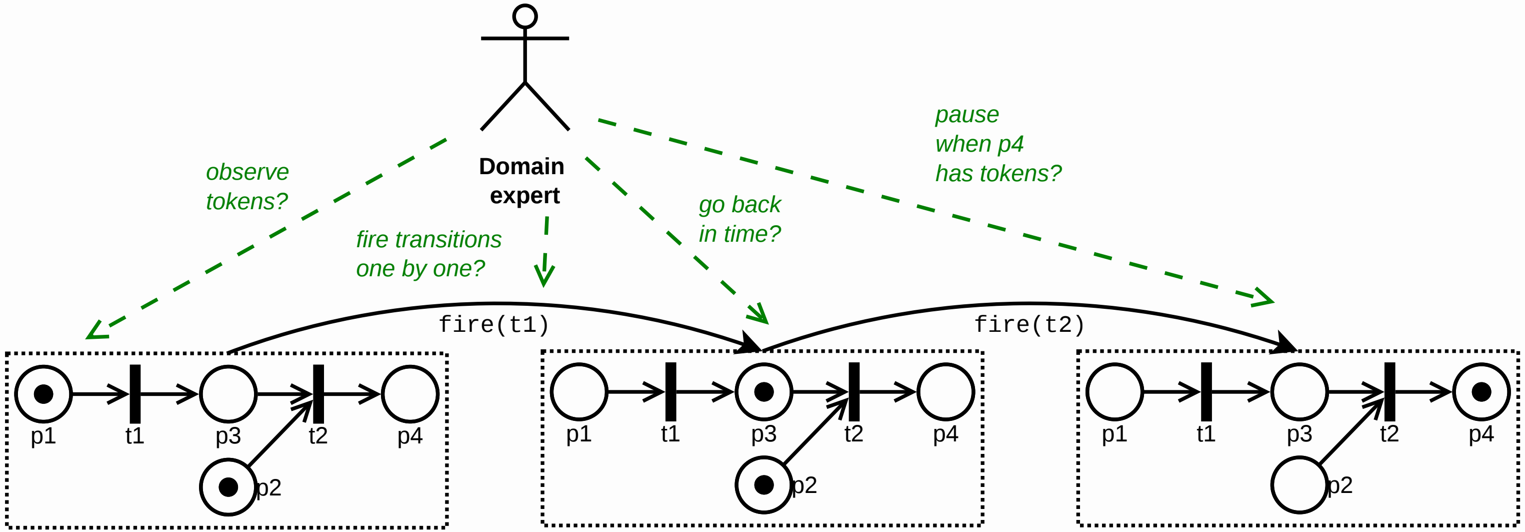
Example of Petri net trace (states only)



Case 2: Generic omniscient debugging

Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, Benoit Baudry. *Omniscient debugging for executable DSLs*. Journal of Systems and Software, 2018.

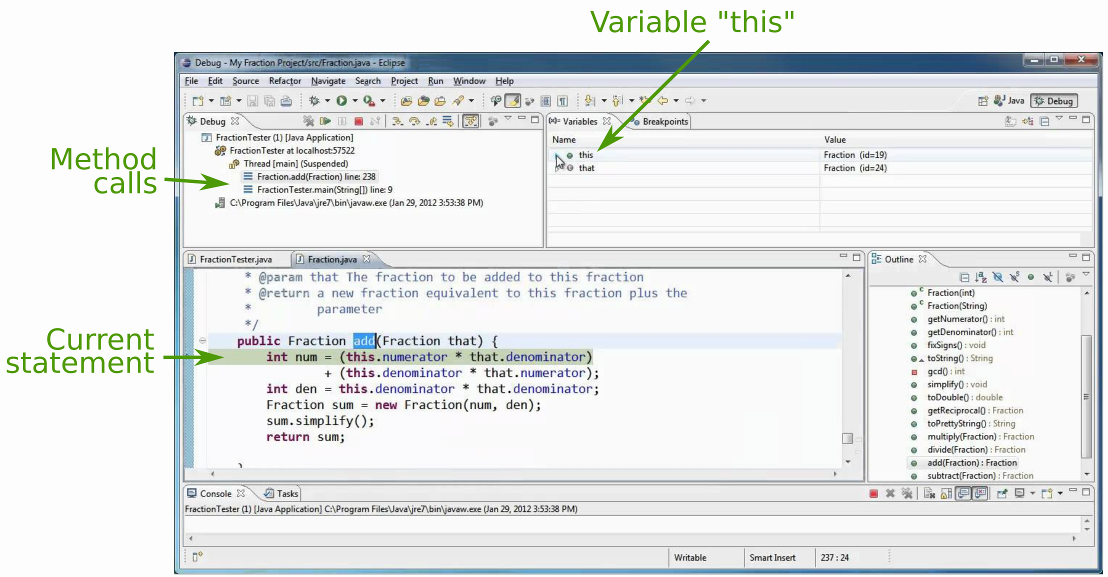
Omniscient debugging



Problem

- *Interactive debugging* is a very common and required service to better understand models through interactive execution and observation facilities
- *Omniscient debugging* extends interactive debugging with facilities to re-explore former execution states during a live execution
- How can we provide omniscient debugging services for any kind of DSL?

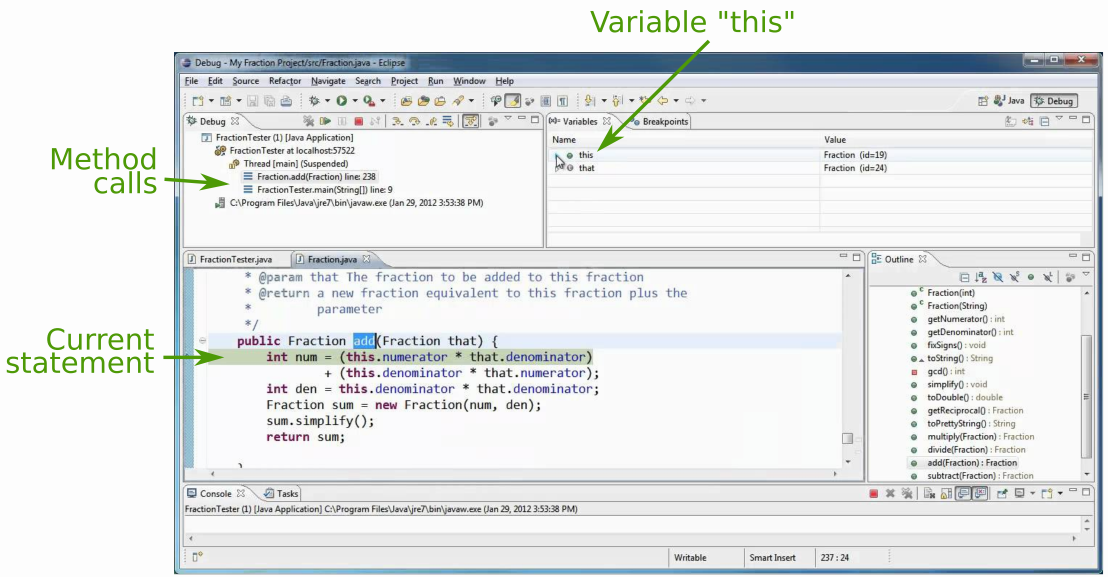
Example of interactive debugging for a specific language



Eclipse JDT Java Debugger

- Debuggers are mostly known for debugging imperative programs
- Concepts such as stack of *method calls*, current *statement*, "this" variable, ...
- Cannot be used to debug other kinds of languages (*eg.* Petri nets)

Example of interactive debugging for a specific language



Eclipse JDT Java Debugger

- Debuggers are mostly known for debugging imperative programs
- Concepts such as stack of *method calls*, current *statement*, "this" variable, ...
- Cannot be used to debug other kinds of languages (*eg.* Petri nets)

We must re-think interactive debugging in a *language-agnostic* fashion

Towards *generic* omniscient debugging

- In software engineering, interactive debugging relies on well-known concepts:
  - a *breakpoint* is a marker that is put on a specific line of code, (or on a method, or an exception), and that will pause the execution once reached
  - the *“step into”* operation means going to the first statement of the next method call,
  - backwards operators (*eg.* “back into” or "play reverse") provide the same services in reverse
  - ...

Towards *generic* omniscient debugging

- In software engineering, interactive debugging relies on well-known concepts:
  - a *breakpoint* is a marker that is put on a specific line of code, (or on a method, or an exception), and that will pause the execution once reached
  - the *“step into”* operation means going to the first statement of the next method call,
  - backwards operators (*eg.* “back into” or "play reverse") provide the same services in reverse
  - ...
- Providing generic interactive debugging for any DSL (*ie.* any context) requires redefining these software engineering concepts in a language-agnostic fashion, such as:
  - a *breakpoint* is a predicate on the execution state,
  - *“step into”* means going to the first execution step enclosed in the next execution step.

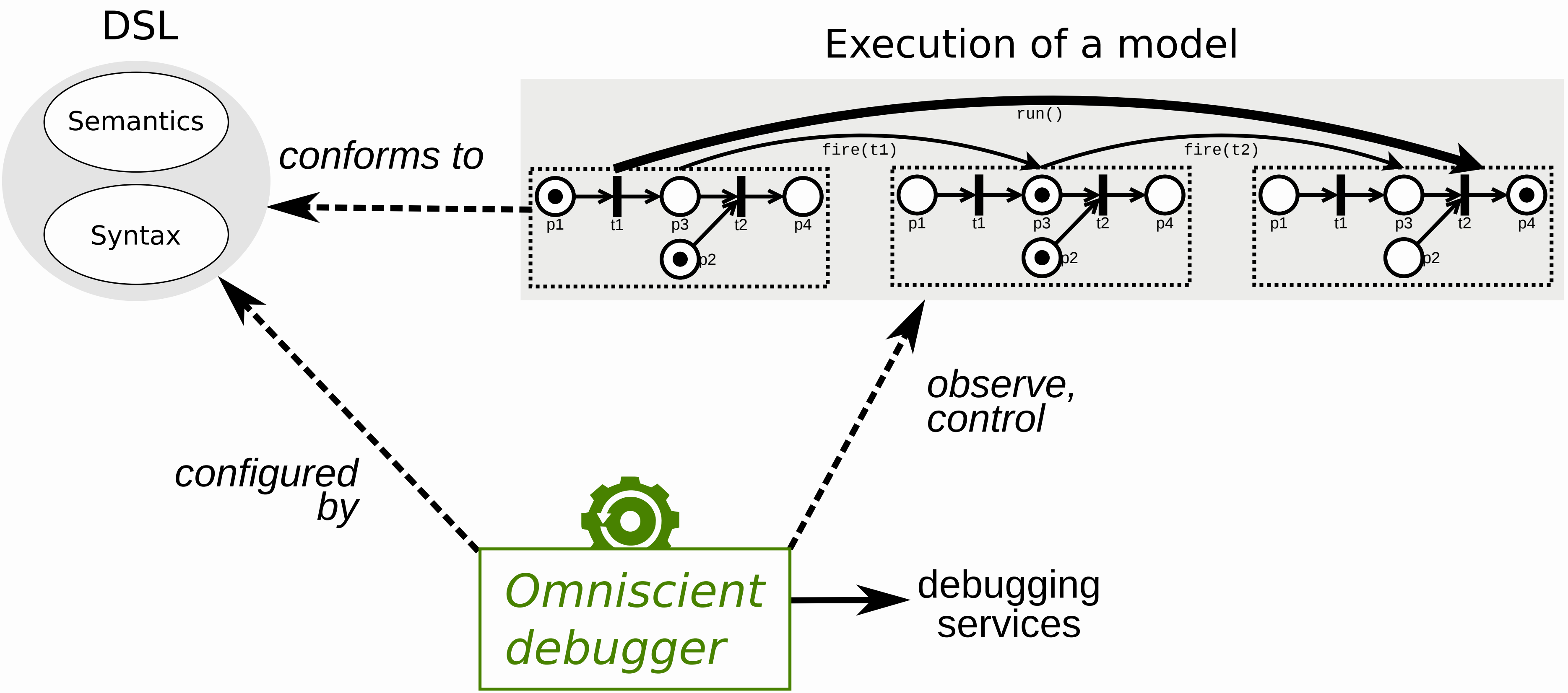
Towards *generic* omniscient debugging

- In software engineering, interactive debugging relies on well-known concepts:
  - a *breakpoint* is a marker that is put on a specific line of code, (or on a method, or an exception), and that will pause the execution once reached
  - the *"step into"* operation means going to the first statement of the next method call,
  - backwards operators (*eg.* "back into" or "play reverse") provide the same services in reverse
  - ...
- Providing generic interactive debugging for any DSL (*ie.* any context) requires redefining these software engineering concepts in a language-agnostic fashion, such as:
  - a *breakpoint* is a predicate on the execution state,
  - *"step into"* means going to the first execution step enclosed in the next execution step.
- Like tracing, requires a DSL enriched with information on execution steps and execution states

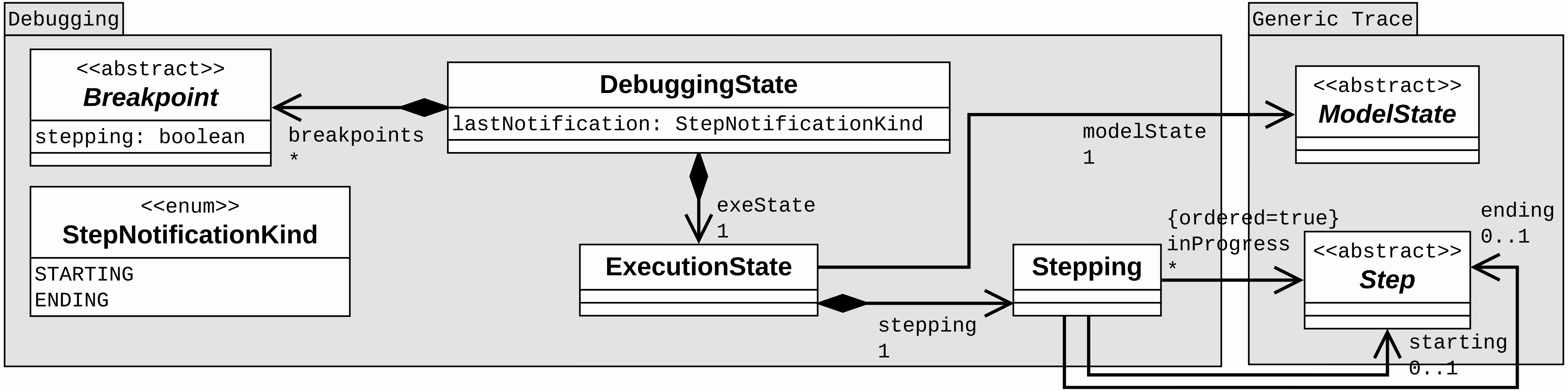
Towards *generic* omniscient debugging

- In software engineering, interactive debugging relies on well-known concepts:
  - a *breakpoint* is a marker that is put on a specific line of code, (or on a method, or an exception), and that will pause the execution once reached
  - the *“step into”* operation means going to the first statement of the next method call,
  - backwards operators (*eg.* “back into” or "play reverse") provide the same services in reverse
  - ...
- Providing generic interactive debugging for any DSL (*ie.* any context) requires redefining these software engineering concepts in a language-agnostic fashion, such as:
  - a *breakpoint* is a predicate on the execution state,
  - *“step into”* means going to the first execution step enclosed in the next execution step.
- Like tracing, requires a DSL enriched with information on execution steps and execution states

Definition of a generic debugger configured with the DSL definition



Generic omniscient debugging metamodel

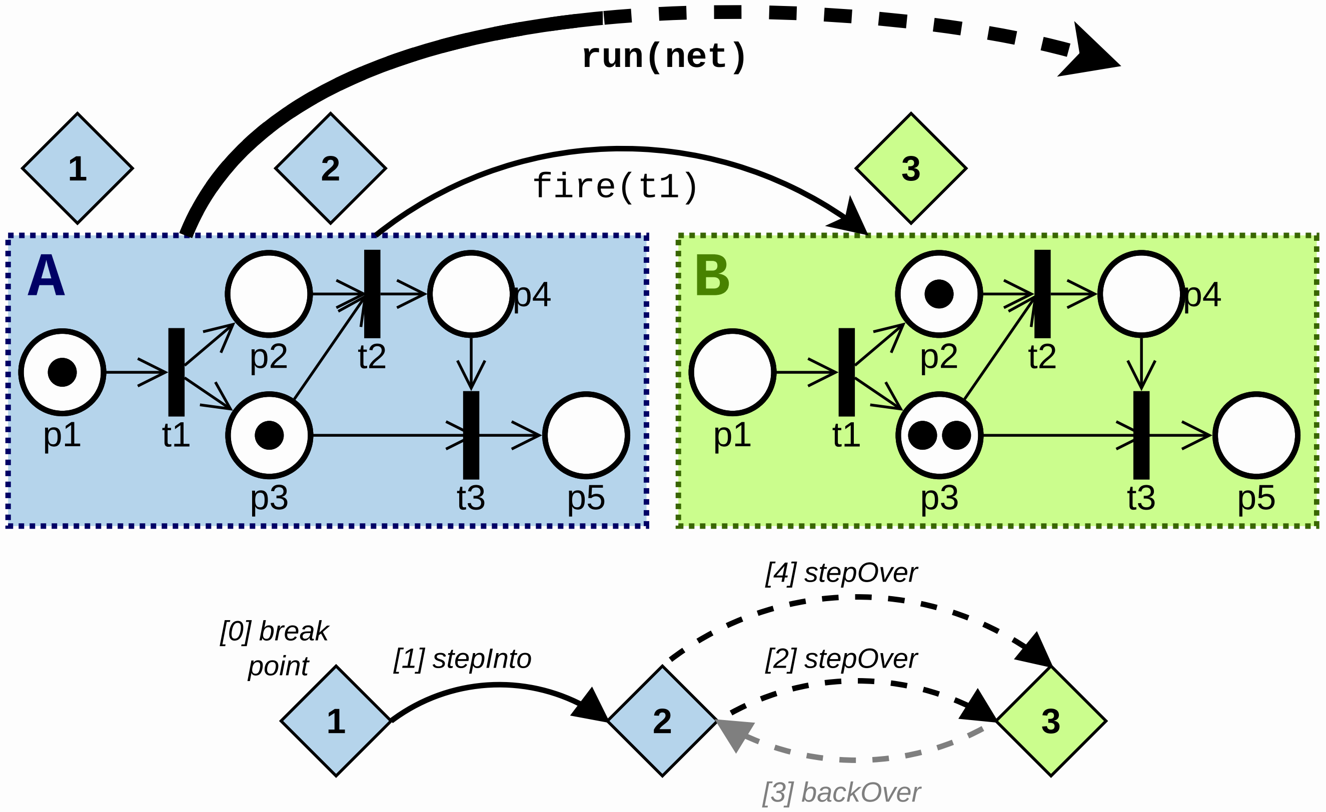


Generic definition of debugging services (excerpt)

stepInto()	**if** $d_{state}.\texttt{exeState.stepping.starting} \neq \mathit{null}$ **then** $d_{state}.\texttt{breakpoints}.\mathrm{add}([\mathit{true}])$ play()
stepOver()	**if** $d_{state}.\texttt{exeState.stepping.starting} \neq \mathit{null}$ **then** $s_{over} \leftarrow d_{state}.\texttt{exeState.stepping.starting}$ $d_{state}.\texttt{breakpoints}.\mathrm{add}([d_{state}.\texttt{exeState.stepping.ending} = s_{over}])$ play()
stepOut()	**if** $d_{state}.\texttt{exeState.stepping.inProgress} \neq \varnothing$ **then** $s_{out} \leftarrow d_{state}.\texttt{exeState.stepping.inProgress}.\mathrm{peek}()$ $d_{state}.\texttt{breakpoints}.\mathrm{add}([d_{state}.\texttt{exeState.stepping.ending} = s_{out}])$ play()

Table 3: Forward services of the omniscient debugger

Example of Petri net generic omniscient debugging



Connecting generic tools to DSLs at runtime

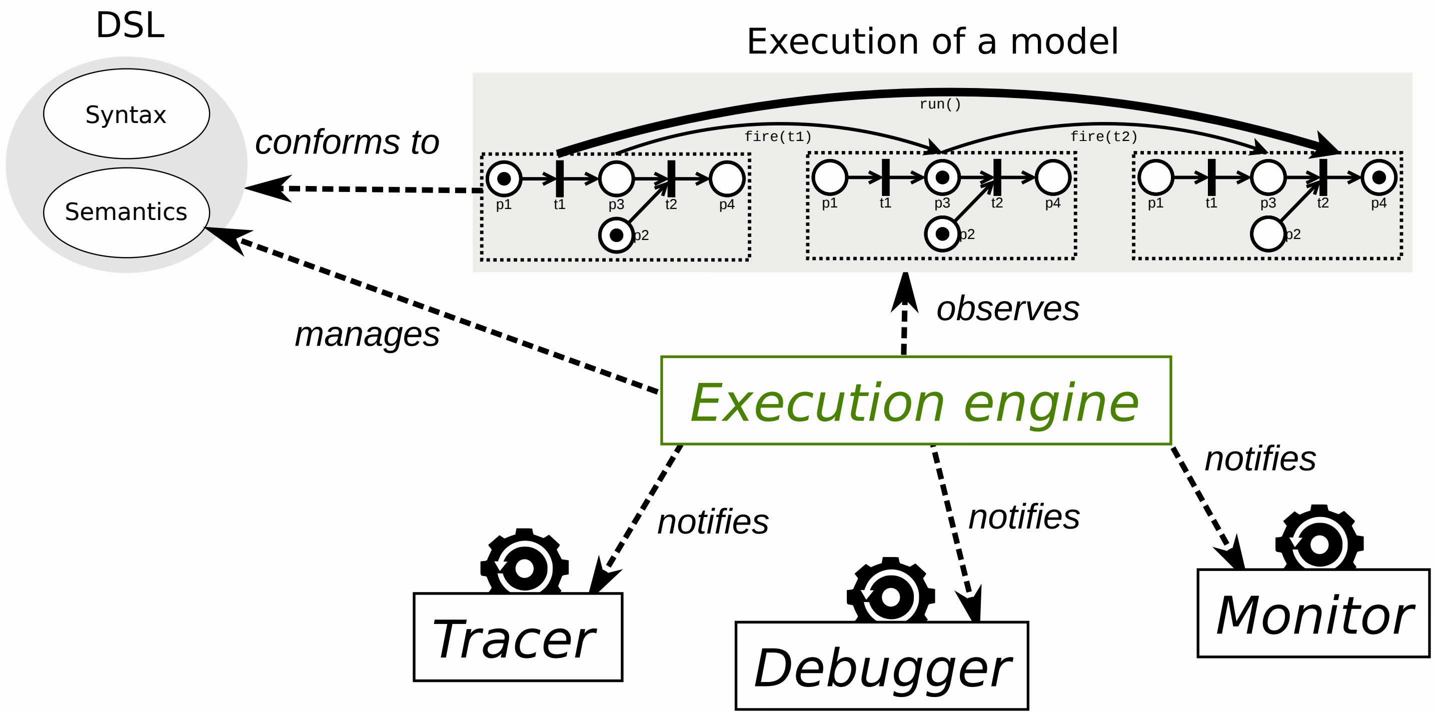
Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, Benoit Combemale. *Execution Framework of the GEMOC Studio (Tool Demo)*. International Conference on Software Language Engineering (SLE), 2016.

Using an execution engine as an intermediary (1)

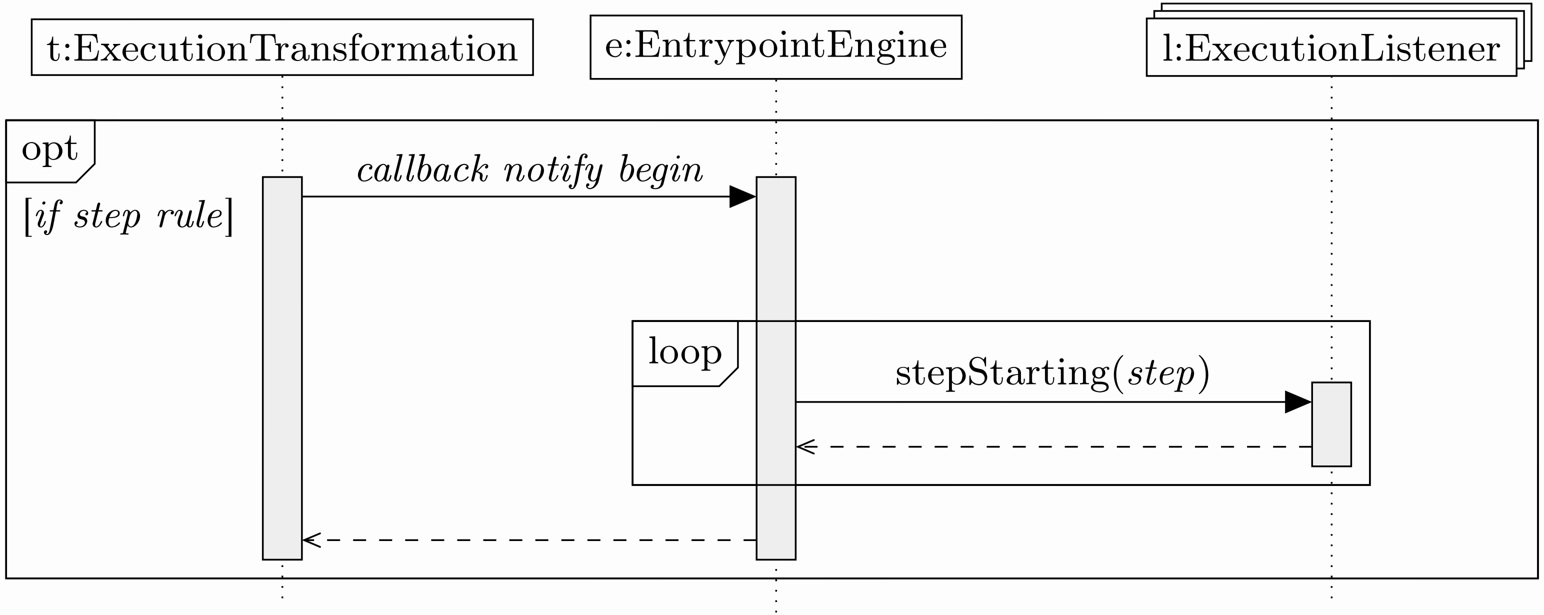
- Problem: once defined or generated, how can a generic tool interact with an executed model?

Using an execution engine as an intermediary (1)

- Problem: once defined or generated, how can a generic tool interact with an executed model?
- To mitigate the intrusiveness of tools, connection of the semantics with a unique execution engine, often through some instrumentation of the interpreter



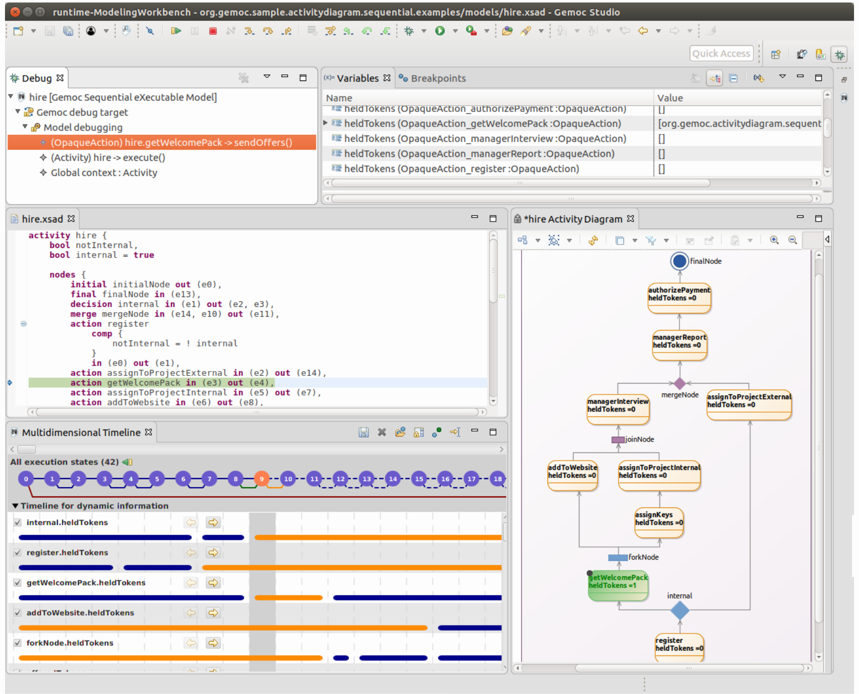
Using an execution engine as an intermediary (2)



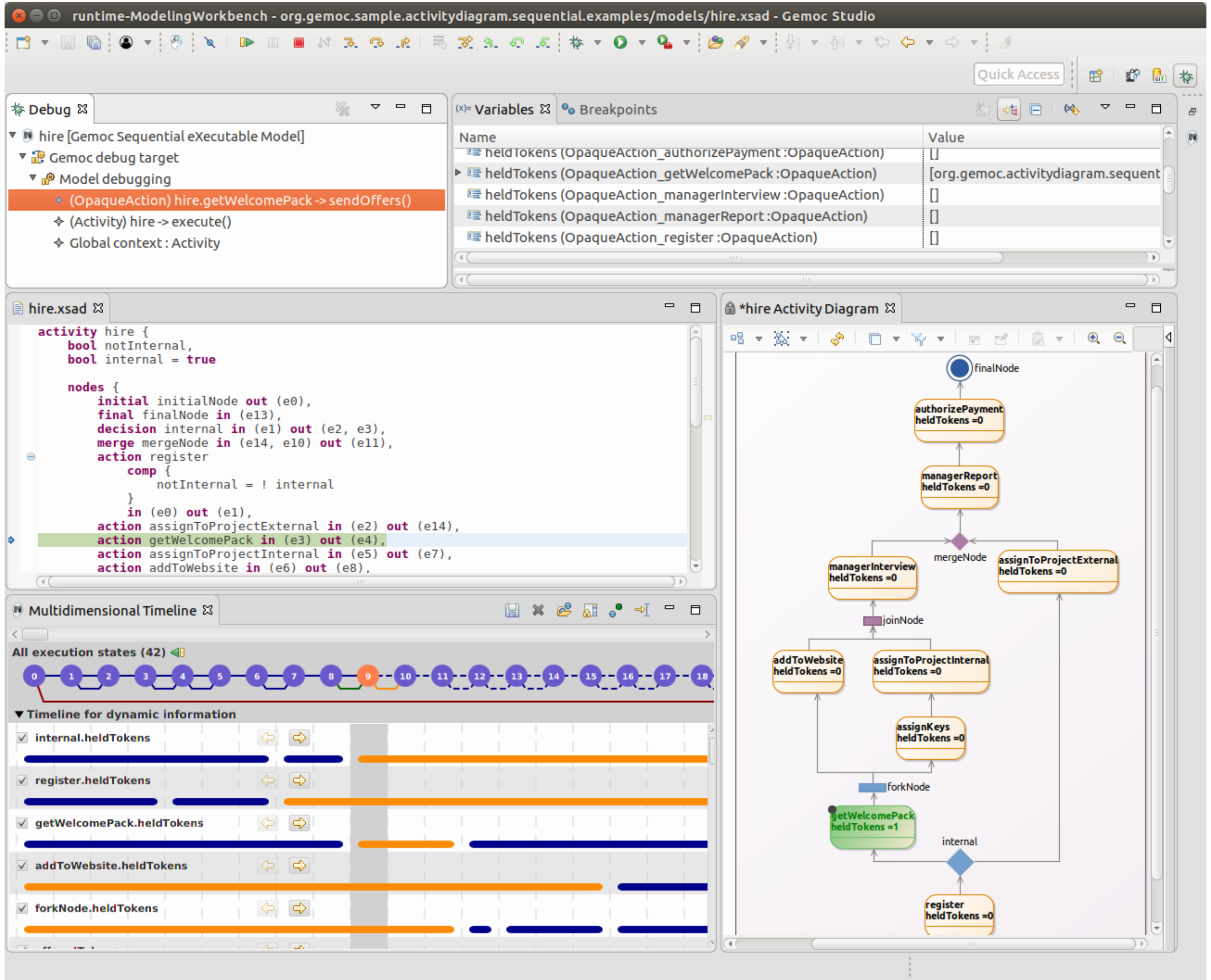
Implementation in the GEMOC Studio

GEMOC Studio

- Open-source Eclipse-based workbench atop the Eclipse Modeling Framework (EMF), in two parts:
  - language workbench: used by language designers to build and compose new executable DSLs,
  - modeling workbench: used by domain designers to create, execute and coordinate models conforming to executable DSLs.
- Handled by the *GEMOC initiative*, an informal group with partners from both the academia and the industry
- Now an official research consortium of the Eclipse foundation



Generic dynamic tools in the GEMOC Studio



Conclusion and Looking forward

Conclusion

Mitigating the tool explosion problem with generic tooling

- DSLs are *central assets* when using MDE to design all aspects of complex systems
- However, tool explosion problem:
  - a system requires a wide range of DSLs,
  - a DSL requires a wide range of tools,
  - thus huge cost development effort.
- Presented mitigation: generic tooling
  - must be well-scoped and language-agnostic
  - often require enriching the DSL definition
  - two examples shown: tracing and omniscient debugging

Conclusion

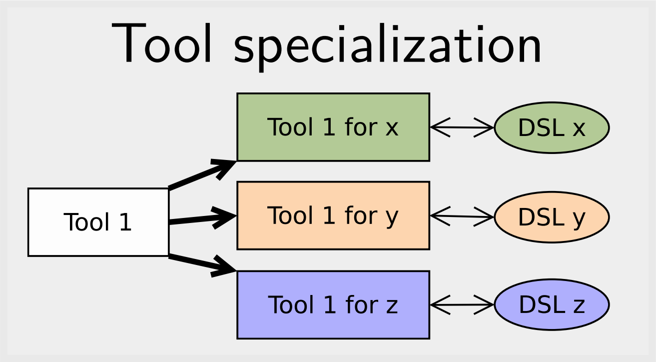
Mitigating the tool explosion problem with generic tooling

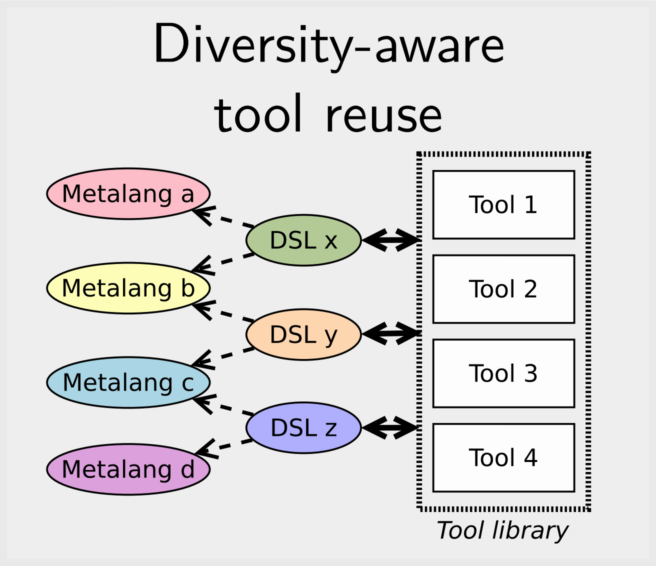
- DSLs are *central assets* when using MDE to design all aspects of complex systems
- However, tool explosion problem:
    - a system requires a wide range of DSLs,
    - a DSL requires a wide range of tools,
    - thus huge cost development effort.
- Presented mitigation: generic tooling
    - must be well-scoped and language-agnostic
    - often require enriching the DSL definition
    - two examples shown: tracing and omniscient debugging

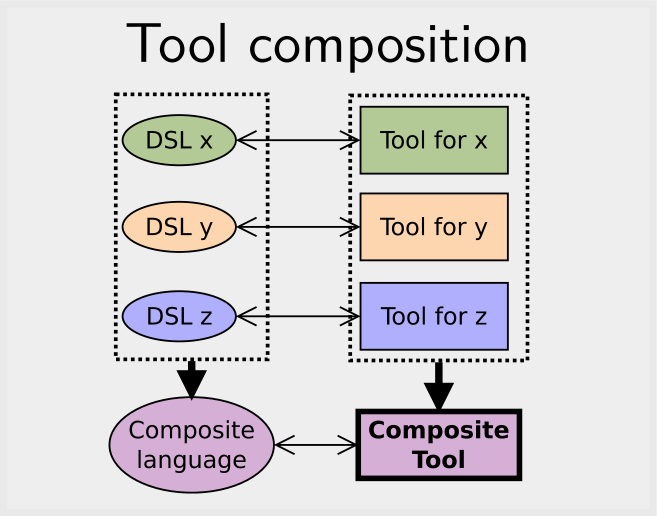
But no silver bullet!

- While saving costs, generic tools cannot compete with handmade finely tuned domain-specific tools
- Extremely useful for new DSLs that have no or little tool-support
- *Possible strategy:* progressively replace generic tools with handmade domain-specific ones

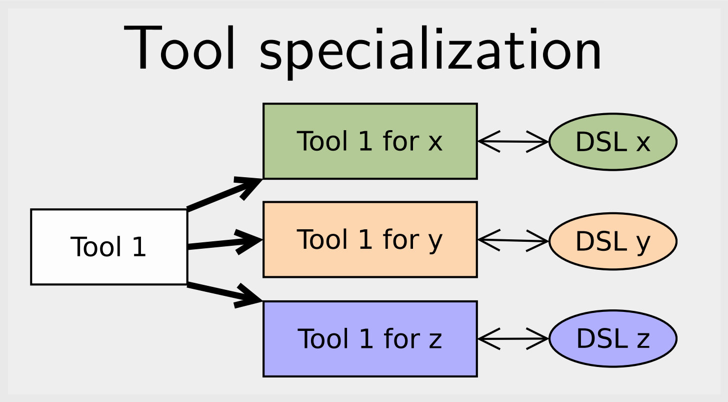
Looking forward







Looking forward (1): better tool specialization

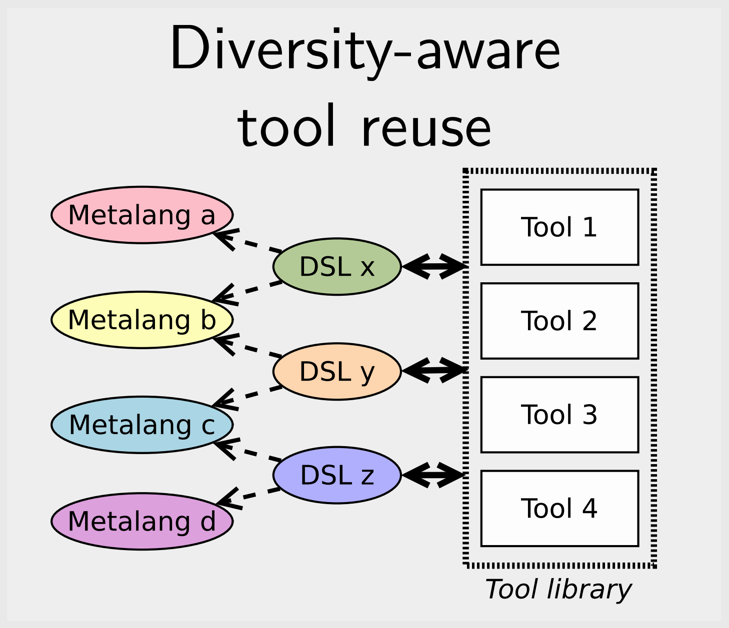


- Generic tools rarely meet the standards of domain-specific tools, as they cannot cover all peculiarities of the domain or needs of domain experts
- Idea: facilitate the *specialization* of a generic tool for a given DSL, which requires:
  - *enriching the DSL* with well chosen extraneous data required for the specialization,
  - *customizing the generic tools* by choosing specific features that may or may not be required for the DSL.

Questions:

- Is is required to *adapt the syntax or semantics* of a DSL to a tool? (eg. `@Step` annotation)
- How to *progressively* specialize a tool, the more enrichment data is provided?

Looking forward (2): mastering DSL diversity

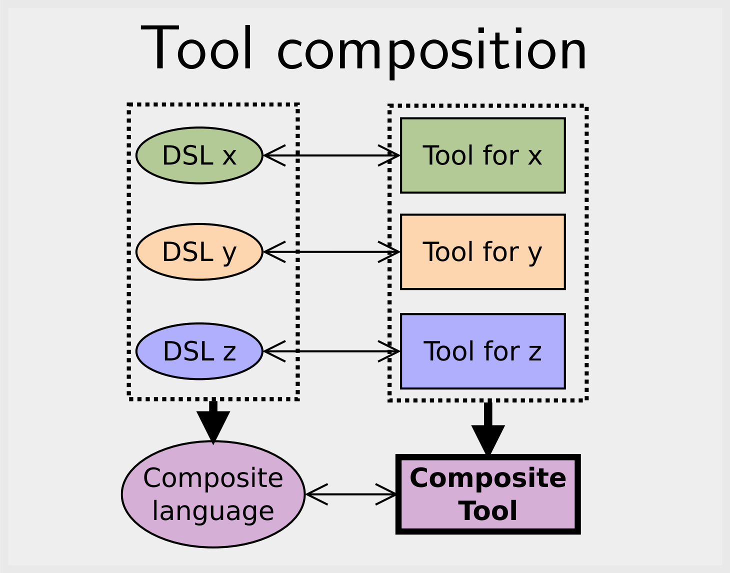


- Huge amount of *diversity*, not only among DSLs (different paradigms, domains, purposes), but also among DSL engineering itself:
  - kinds of abstract syntaxes (metamodel, ADT, etc.)
  - kinds of semantics (operational, translational, rewriting rules, etc.)
  - kinds of concrete syntaxes (graphical, textual, etc.)
  - used patterns (visitor based interpreted, etc.)
  - used metalanguages (Ecore, Monticore, Kermeta, Rascal, ATL, Spoofax, Coq etc.)

Questions:

- How to make tools that can be reused over a wide scope of DSLs and metalanguages?
- At runtime, how to deal with all sorts of semantics?
- Are protocols the future, similarly to the *Language Server Protocol (LSP)*?

Looking forward (3): tool composition



- Wide range of approaches aiming to compose DSLs and/or reuse parts of DSLs
- What about tools? Maybe they should be composed too!
- Questions:
  - Can the composition of tools be derived from the composition of DSLs?
  - Should a composite tool be a "common denominator", or can it benefit from the specificities of each tool?

Main references for this talk

1. Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, Benoit Baudry. *Omniscient debugging for executable DSLs*. Journal of Systems and Software, 2018.
2. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, Benoit Baudry. *Advanced and efficient execution trace management for executable domain-specific modeling languages*. Software & Systems Modeling, 2017.
3. Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, Benoit Combemale. *Execution Framework of the GEMOC Studio (Tool Demo)*. International Conference on Software Language Engineering (SLE), 2016.
4. Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, Benoit Baudry. *Supporting Efficient and Advanced Omniscient Debugging for xDSMLs.* International Conference on Software Language Engineering (SLE), 2015.
5. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, Benoit Baudry. A *Generative Approach to Define Rich Domain-Specific Trace Metamodels.* European Conference on Modeling Foundations and Applications (ECMFA), 2015.

Thank you for your attention!

https://bousse-e.univ-nantes.io