# Execution Trace Management to Support Dynamic V&V for Executable DSMLs

## Ph.D defense

December 3, 2015

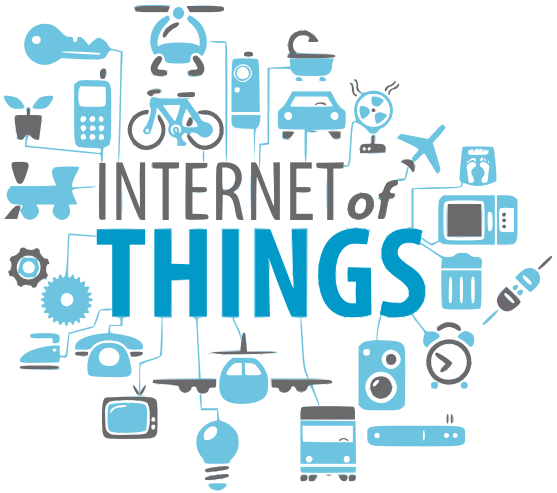
### Jury

Prof. Gerti Kappel, TU Wien / *rapporteur*
Prof. Franck Barbier, UPPA / *rapporteur*
Dr. Julien DeAntoni, UNS / *examinateur*
Prof. François Taïani, UR1 / *examinateur*
Dr. Benoit Baudry, Inria / *directeur de thèse*
Dr. Benoit Combemale, Inria – UR1 / *co-encadrant*

Erwan Bousse
University of Rennes 1

# Increasing complexity of systems

AIRBUS A380

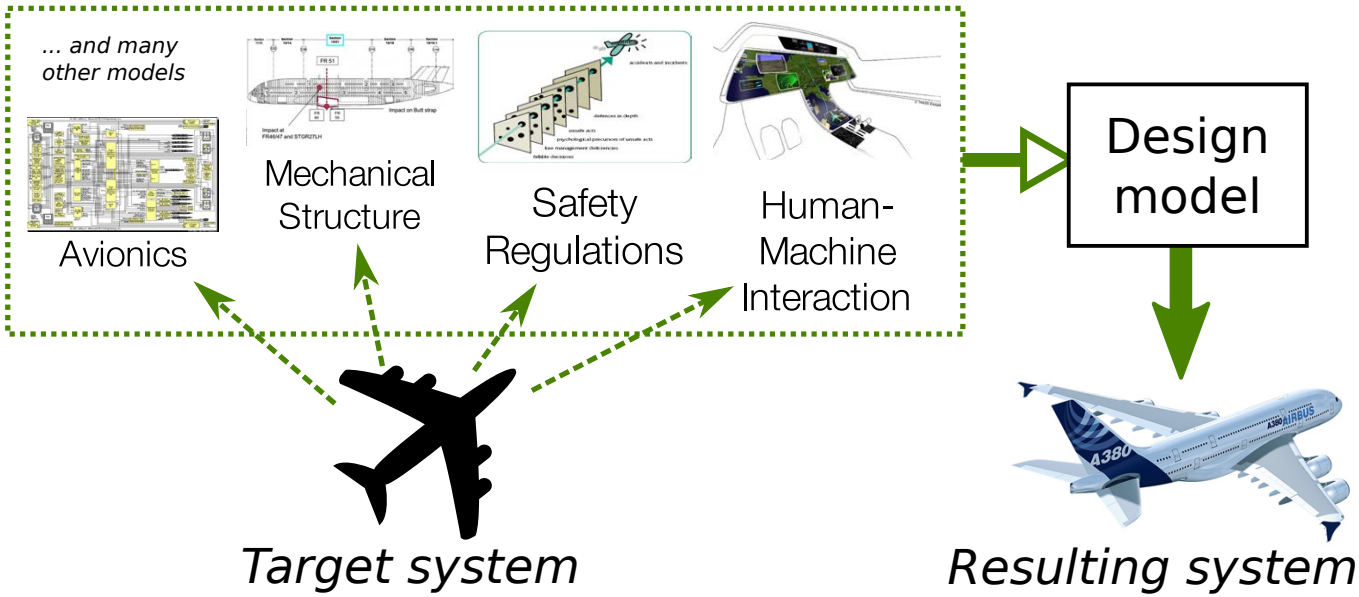



CCP EVE ONLINE

- **Systems are more and more complex**: cyber physical systems, internet of things, massively multiplayer games, ...
- One system = **diverse and heterogeneous domains**

$\Rightarrow$ Many threats to their proper development and functioning

# Model driven engineering (MDE)




- **Separation of concerns** through the use of **models**
- Using **domain specific modeling languages** (DSMLs)
- Enables **early verification and validation** (early V&V)

# Dynamic V&V of behavioral models

- **Behavioral models** are essential in various domains and in various forms of engineering
- **Dynamic V&V** required to check their behavioral properties
  - requires models to be **executable**
  - requires defining the **execution semantics** of the DSMLs
- **Execution semantics** are composed of
  - definition of the **execution state** of conforming models
  - definition of the **execution function** that changes this state

DSML + execution semantics = **executable DSML** (xDSML)

# Dynamic V&V of behavioral models

- **Behavioral models** are essential in various domains and in various forms of engineering
- **Dynamic V&V** required to check their behavioral properties
  - requires models to be **executable**
  - requires defining the **execution semantics** of the DSMLs

- **Execution semantics** are composed of
  - definition of the **execution state** of conforming models
  - definition of the **execution function** that changes this state

DSML + execution semantics = **executable DSML** (xDSML)

# Dynamic V&V of behavioral models

- **Behavioral models** are essential in various domains and in various forms of engineering
- **Dynamic V&V** required to check their behavioral properties
  - requires models to be **executable**
  - requires defining the **execution semantics** of the DSMLs

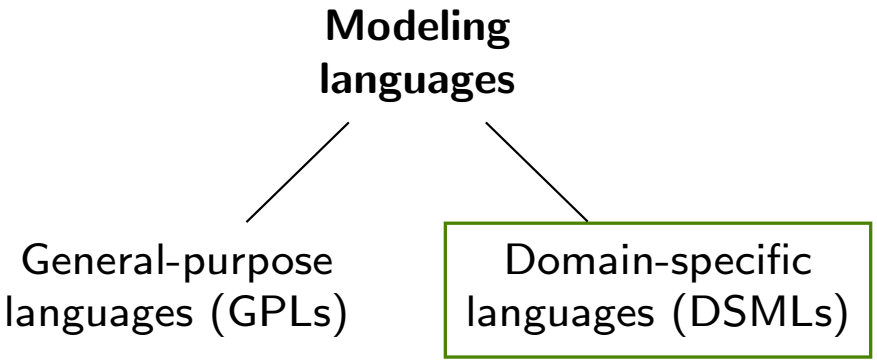
- **Execution semantics** are composed of
  - definition of the **execution state** of conforming models
  - definition of the **execution function** that changes this state

DSML + execution semantics = **executable DSML** (xDSML)

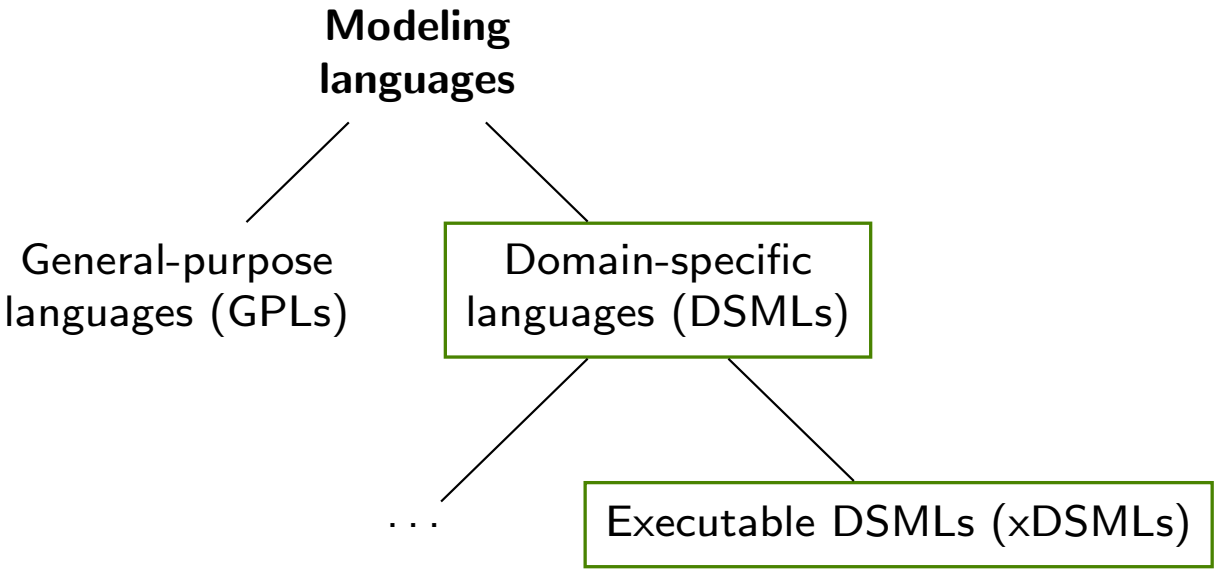
# Scope and hypotheses

**Modeling languages**

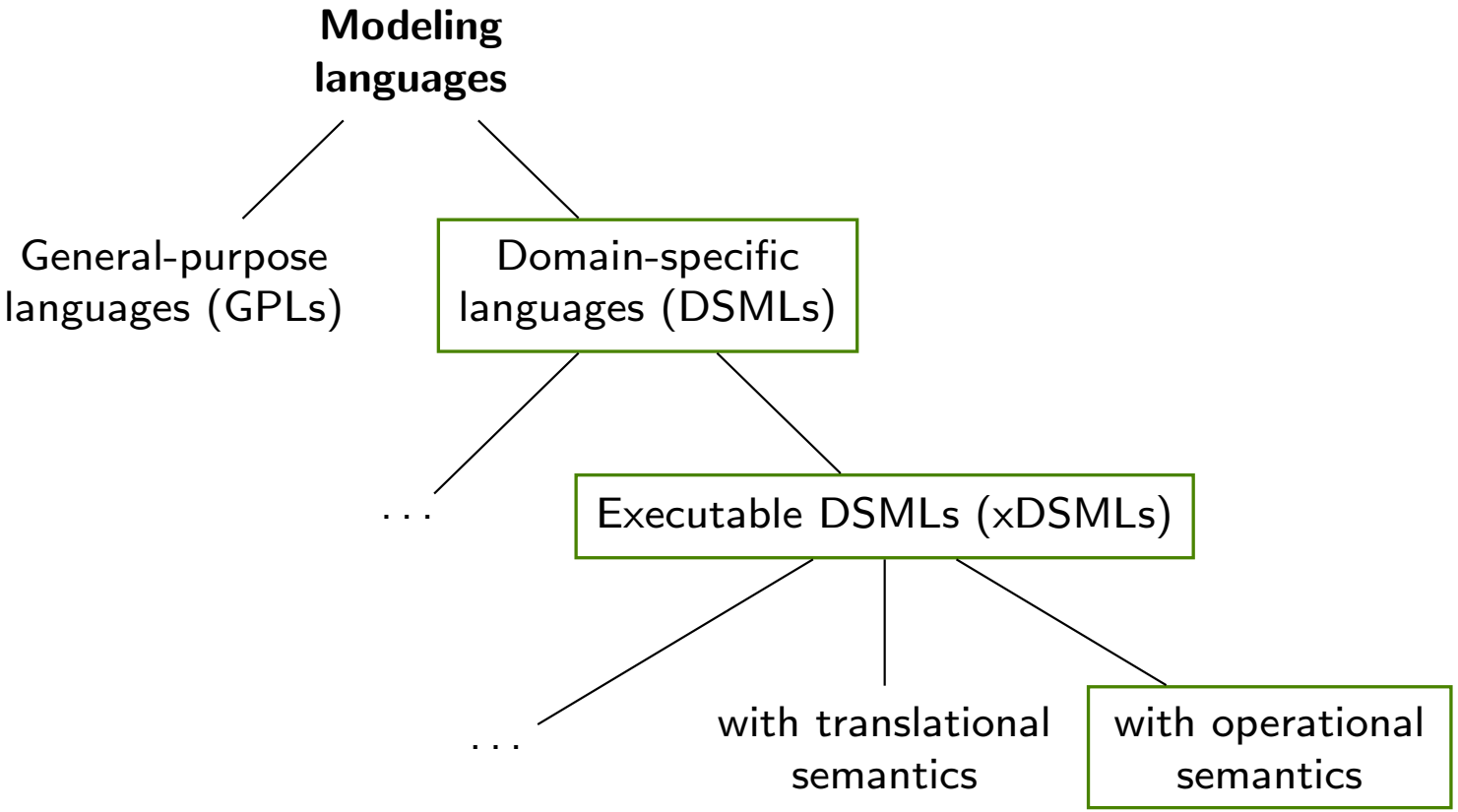
# Scope and hypotheses



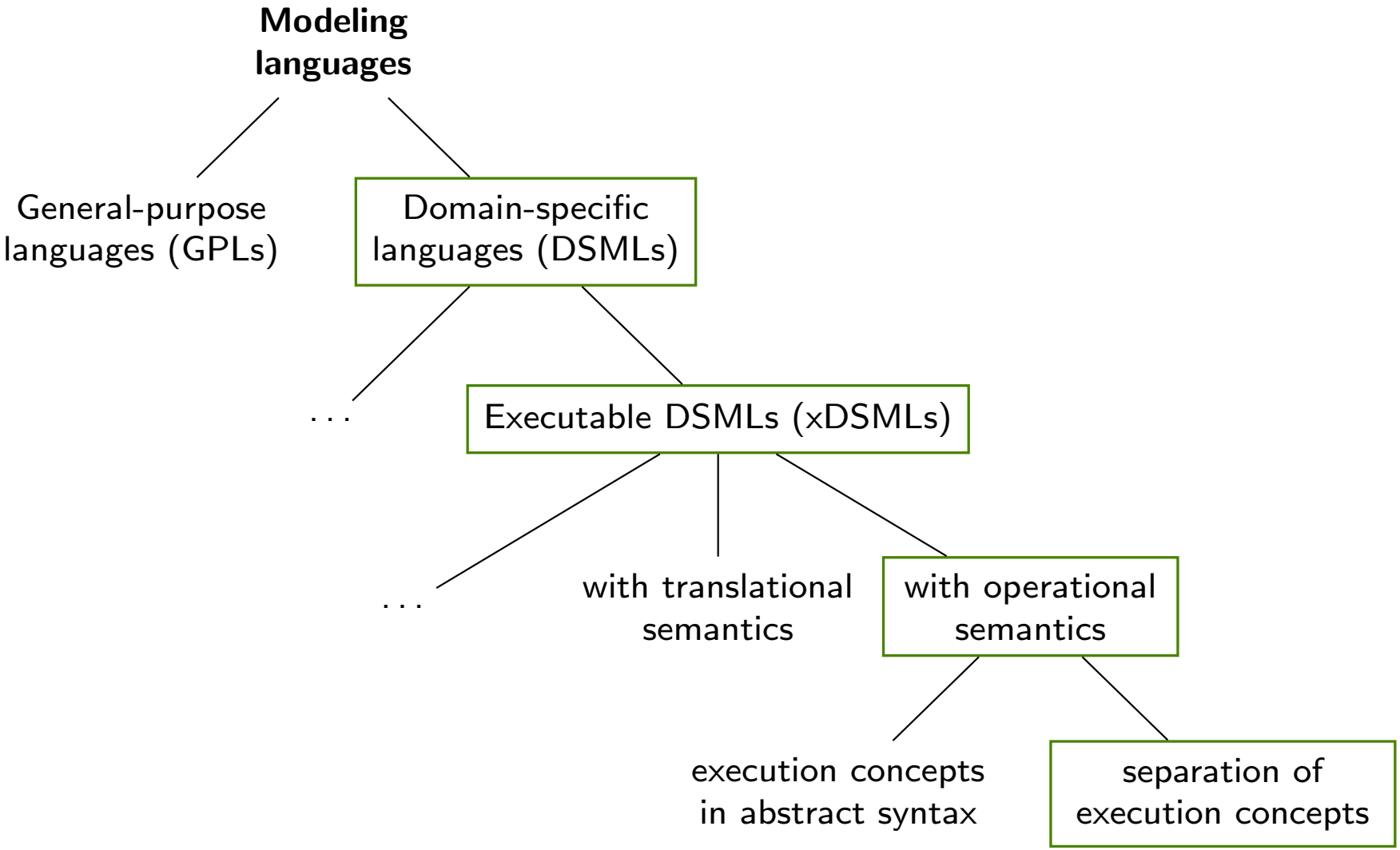
# Scope and hypotheses



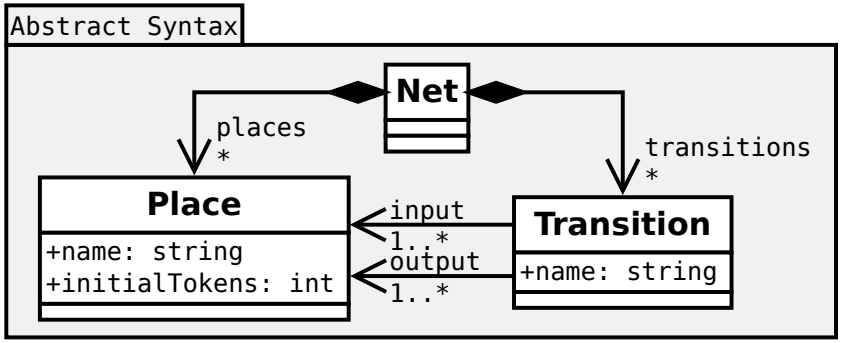
# Scope and hypotheses



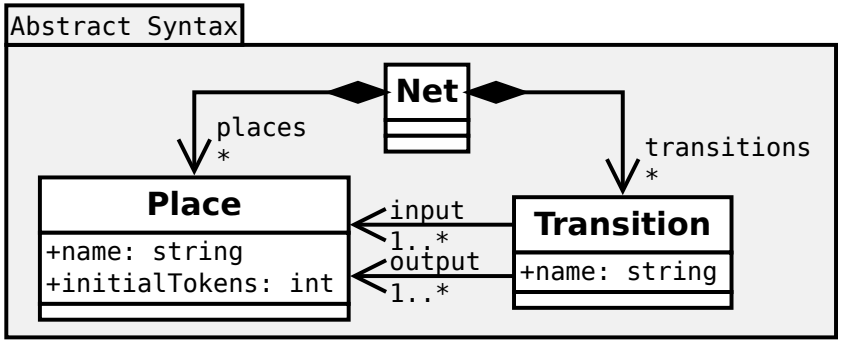
# Scope and hypotheses



# Example of Petri net xDSML and model

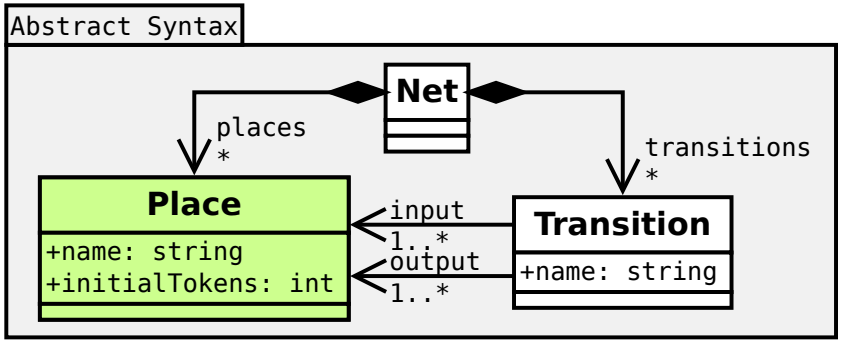


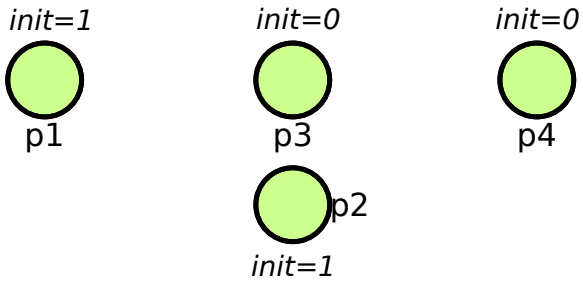
# Example of Petri net xDSML and model




Petri net model

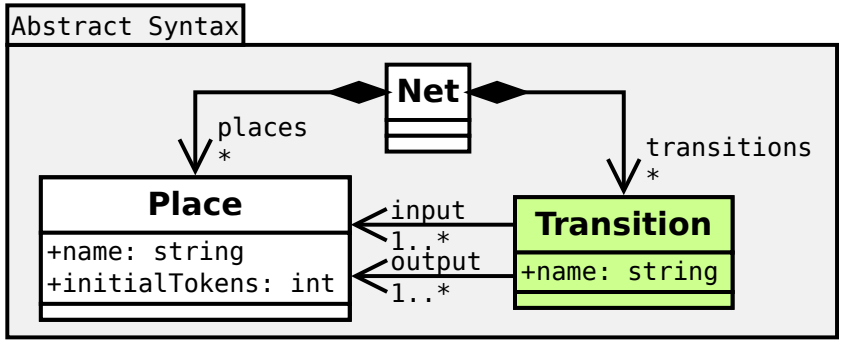
# Example of Petri net xDSML and model

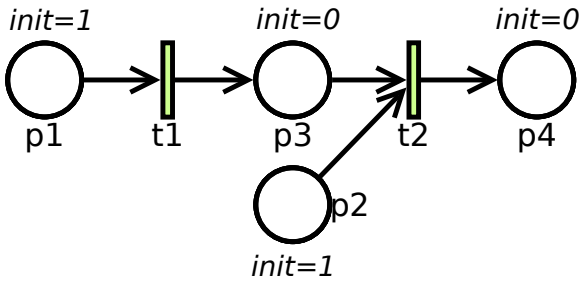



Petri net model

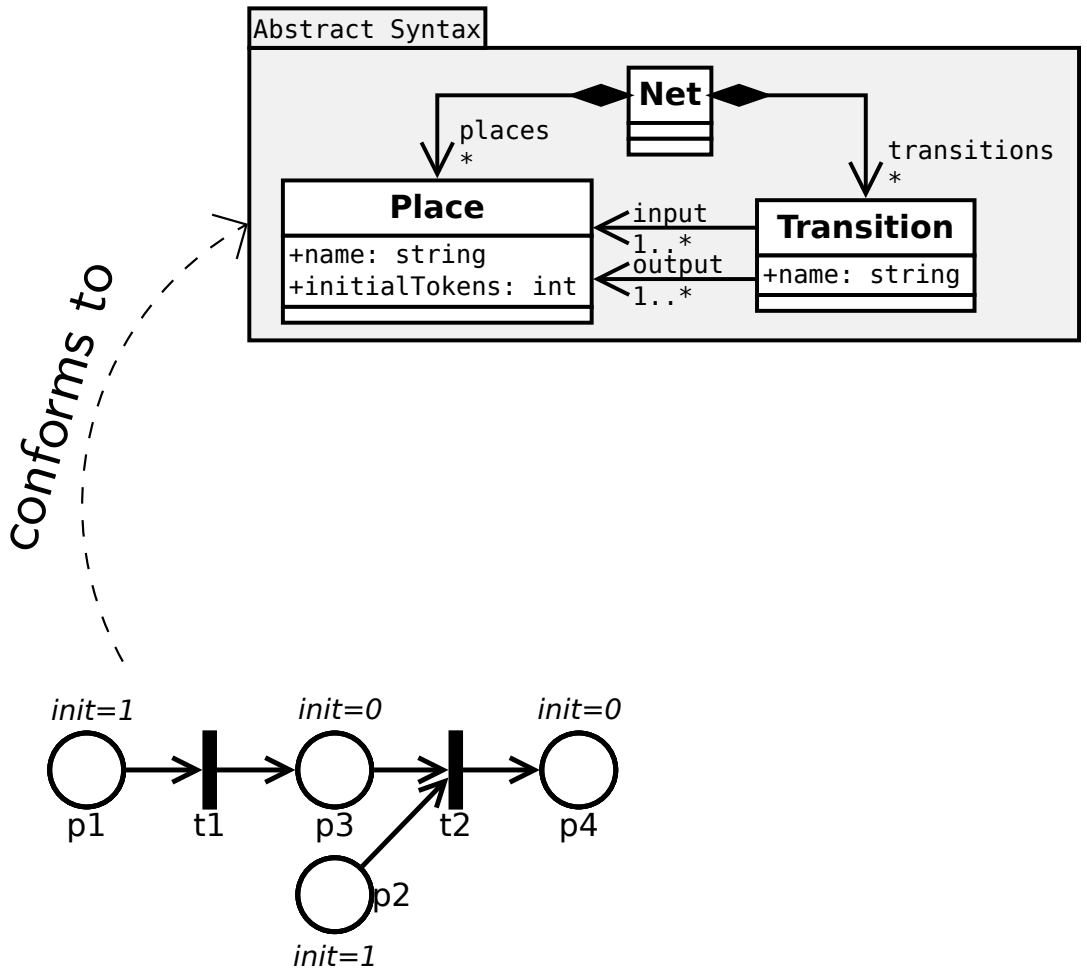
# Example of Petri net xDSML and model



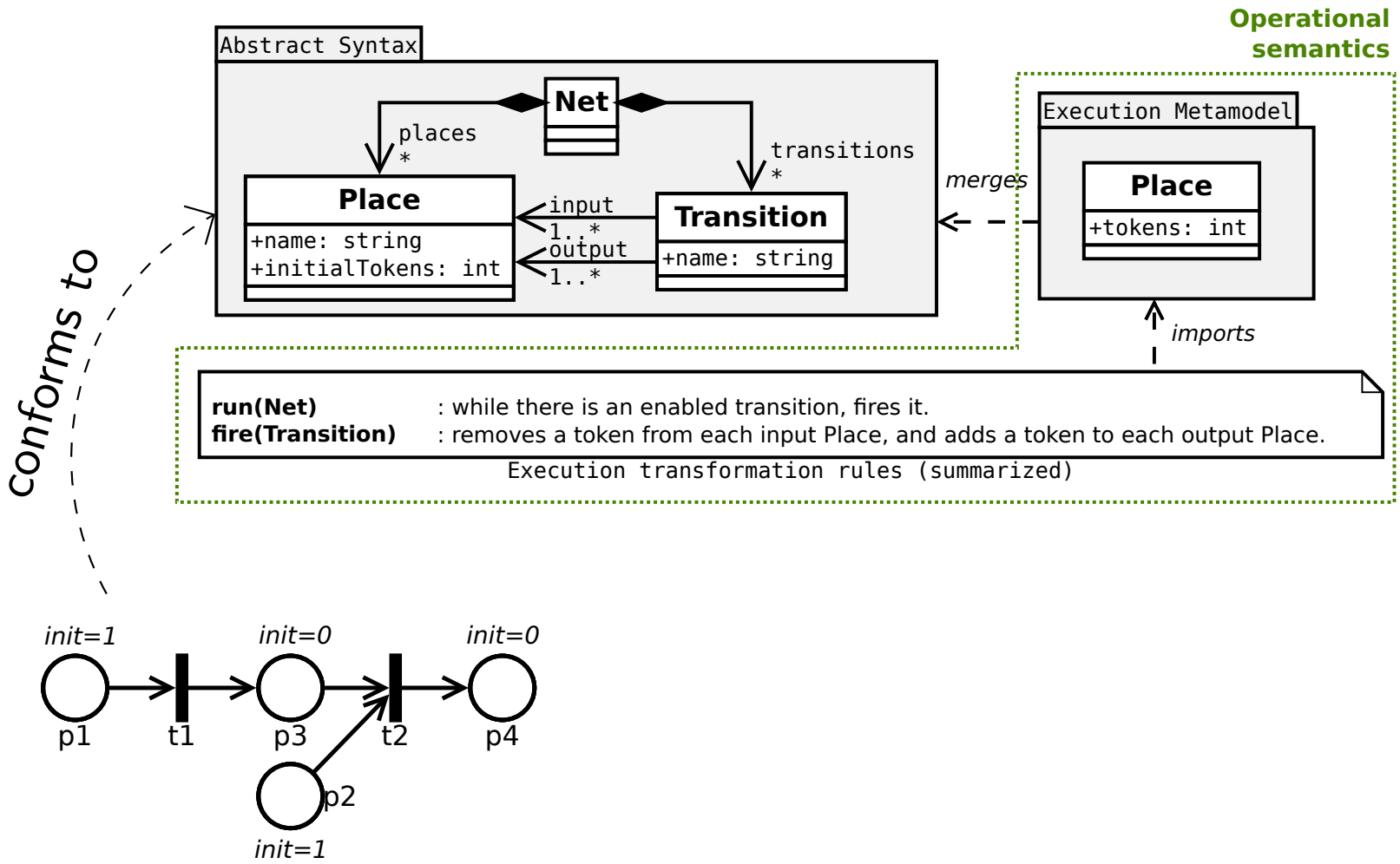

Petri net model

# Example of Petri net xDSML and model



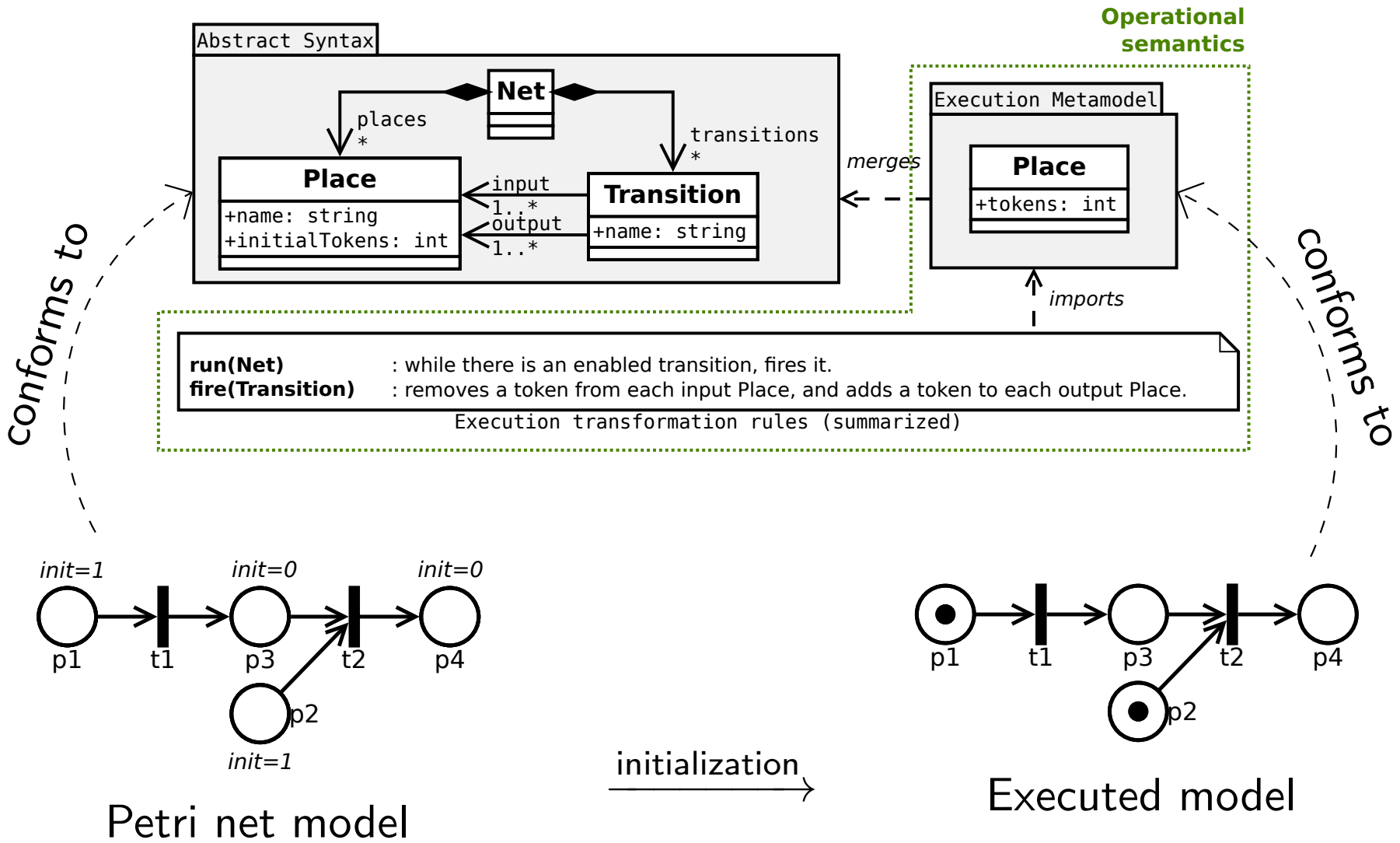

Petri net model

# Example of Petri net xDSML and model

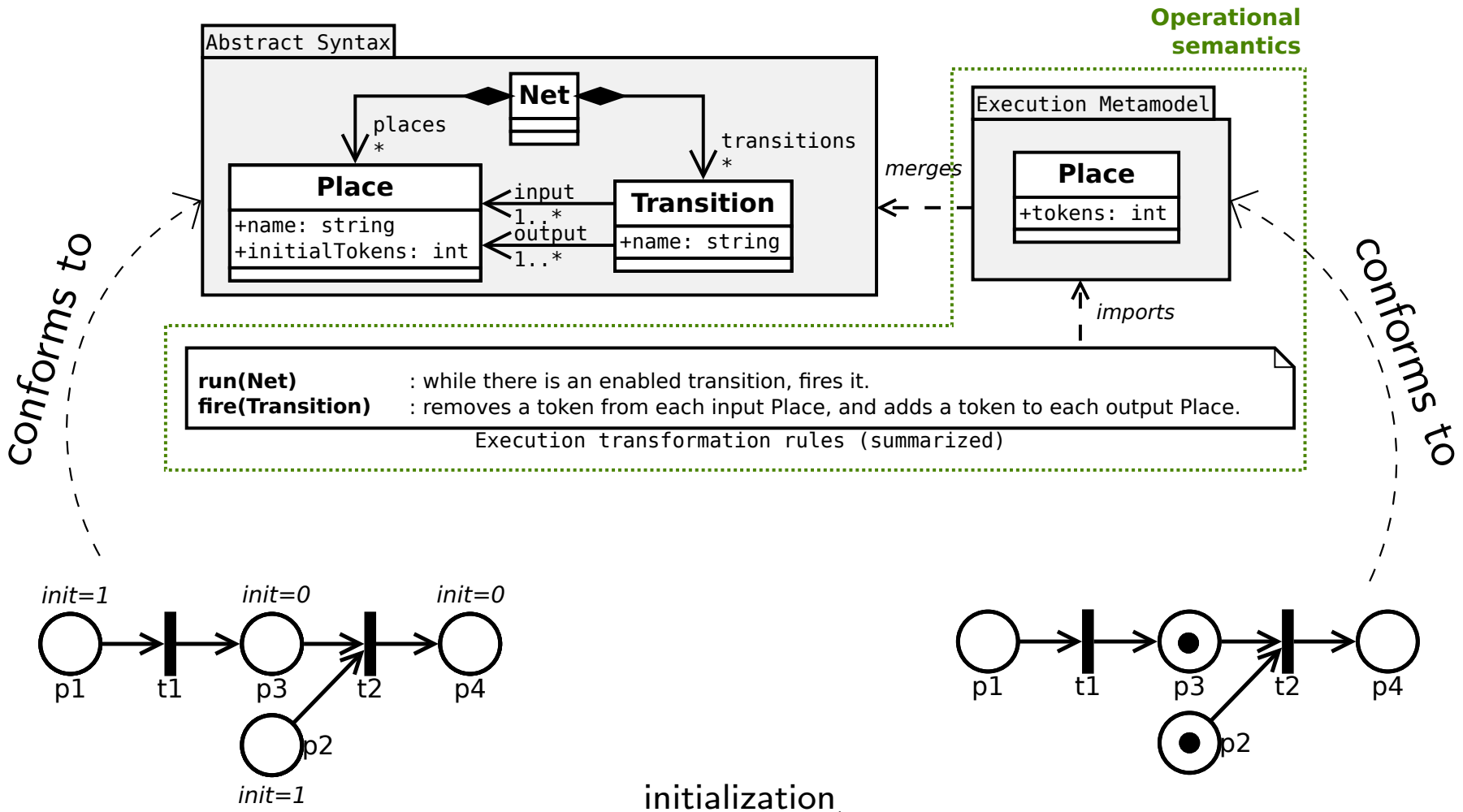



Petri net model

# Example of Petri net xDSML and model



# Example of Petri net xDSML and model

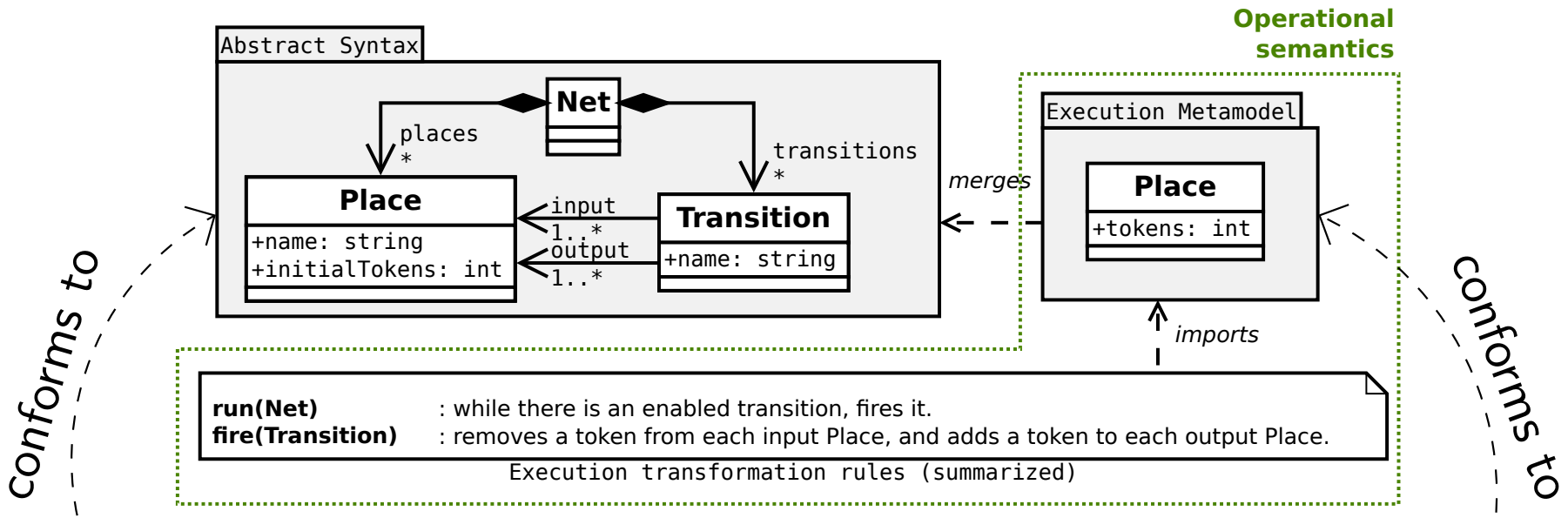


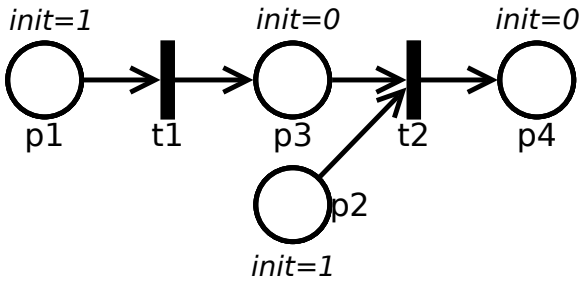

Petri net model

Executed model

*(t1 fired)*

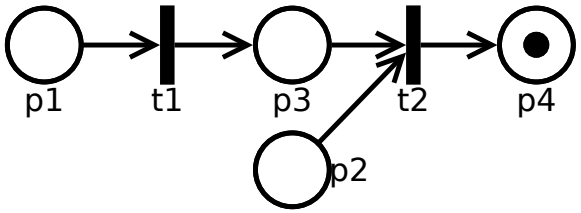
# Example of Petri net xDSML and model




Petri net model

initialization




Executed model

*(t1 fired) (t2 fired)*

# Representing executions as traces

How to **represent executions** in order to **analyze** them?

Example of a Petri net **execution trace**:

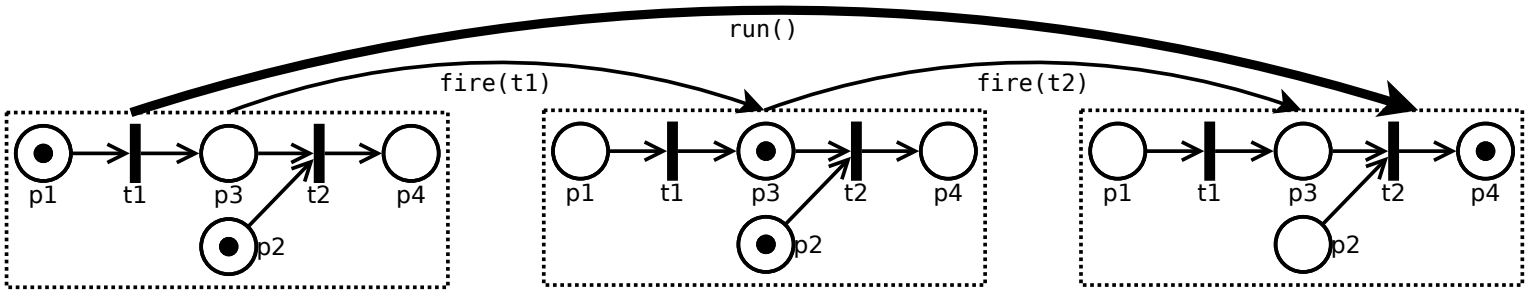



- **Execution states**, each storing the values of the mutable parts of the model (e.g. `tokens`)
- **Execution steps**, each being the application of a *transformation rule* of the operational semantics, with:
  - *small steps* in between execution states (e.g. `fire`)
  - *big steps* containing multiple steps (e.g. `run`)

# Representing executions as traces

How to **represent executions** in order to **analyze** them?

Example of a Petri net **execution trace**:




- **Execution states**, each storing the values of the mutable parts of the model (e.g. `tokens`)
- **Execution steps**, each being the application of a *transformation rule* of the operational semantics, with:
  - *small steps* in between execution states (e.g. `fire`)
  - *big steps* containing multiple steps (e.g. `run`)

# Difficulties in trace management

- **Execution traces can be large**
  - Requires **large amounts of memory** to construct a trace
  - Impacts processing time during analyses
  - *example:* looping small petri net (4 places and 3 transitions), 150 000 states = 130 MB
- **Many kinds of trace manipulations**
  - Generic (e.g. amount of steps)
  - Domain-specific (e.g. number of tokens that traversed a place)
- **Many possible xDSMLs**
  - Existing ones (e.g. Petri net, Activity diagrams)
  - Future ones (cf. language engineering)

# Difficulties in trace management

- **Execution traces can be large**
  - Requires **large amounts of memory** to construct a trace
  - Impacts processing time during analyses
  - *example:* looping small petri net (4 places and 3 transitions), 150 000 states = 130 MB
- **Many kinds of trace manipulations**
  - Generic (e.g. amount of steps)
  - Domain-specific (e.g. number of tokens that traversed a place)
- **Many possible xDSMLs**
  - Existing ones (e.g. Petri net, Activity diagrams)
  - Future ones (cf. language engineering)

# Difficulties in trace management

- **Execution traces can be large**
  - Requires **large amounts of memory** to construct a trace
  - Impacts processing time during analyses
  - *example:* looping small petri net (4 places and 3 transitions), 150 000 states = 130 MB
- **Many kinds of trace manipulations**
  - Generic (e.g. amount of steps)
  - Domain-specific (e.g. number of tokens that traversed a place)
- **Many possible xDSMLs**
  - Existing ones (e.g. Petri net, Activity diagrams)
  - Future ones (cf. language engineering)

# Problem statement

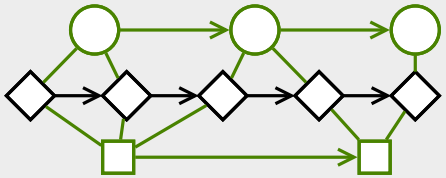
How to provide efficient execution trace management facilities for any xDSML?

Three main inter-related challenges:

1. **Scalability in memory** to construct large traces
2. **Scalability in time** to process large traces
3. **Usability** of the execution trace data structure, to cope with the complexity of data of arbitrarily complex xDSMLs

# Contributions and applications

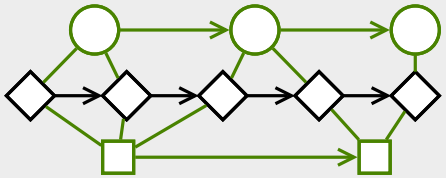
Multidimensionnal domain-specific trace metamodel generation [ECMFA'15]



Scalable armies of model clones [MODELS'14]

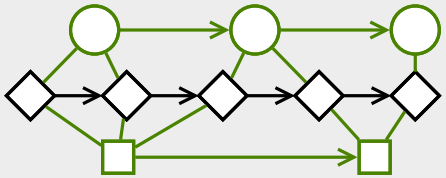
# Contributions and applications

Multidimensionnal domain-specific trace metamodel generation [ECMFA'15]



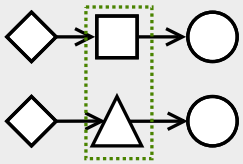
# Contributions and applications

Multidimensionnal domain-specific trace metamodel generation [ECMFA'15]



*applied to*

Enhanced semantic differencing



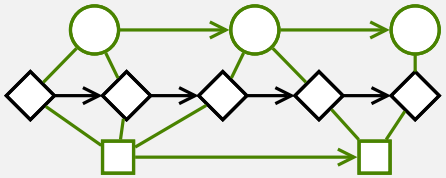
*applied to*

Advanced and efficient omniscient debugging [SLE'15]

# Outline

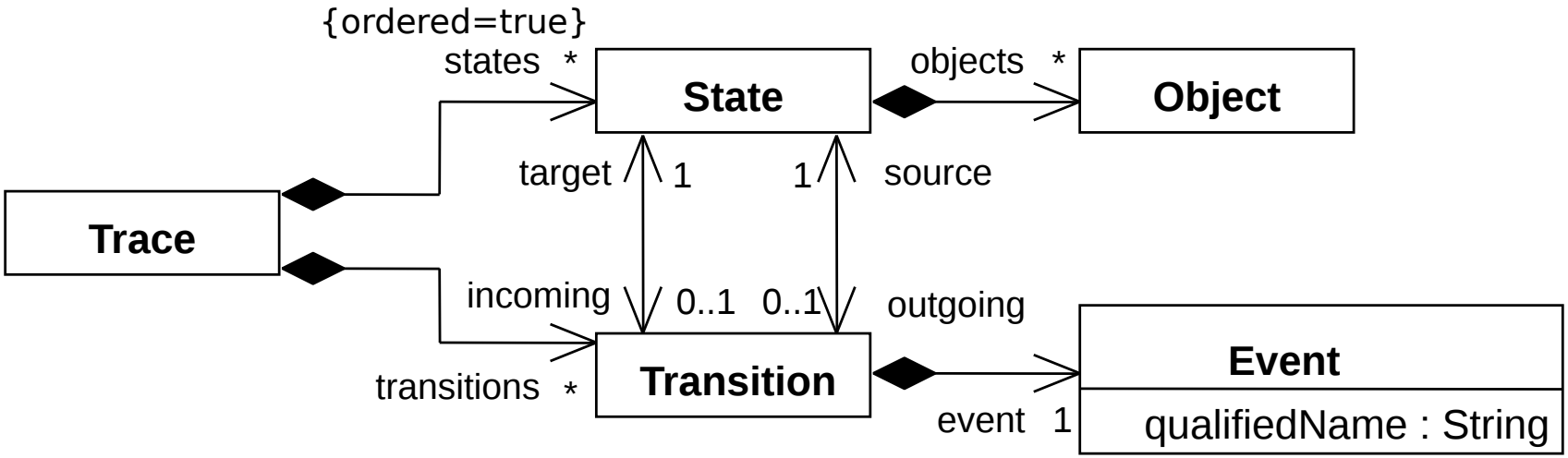


Multidimensionnal domain-specific trace metamodel generation [ECMFA'15]



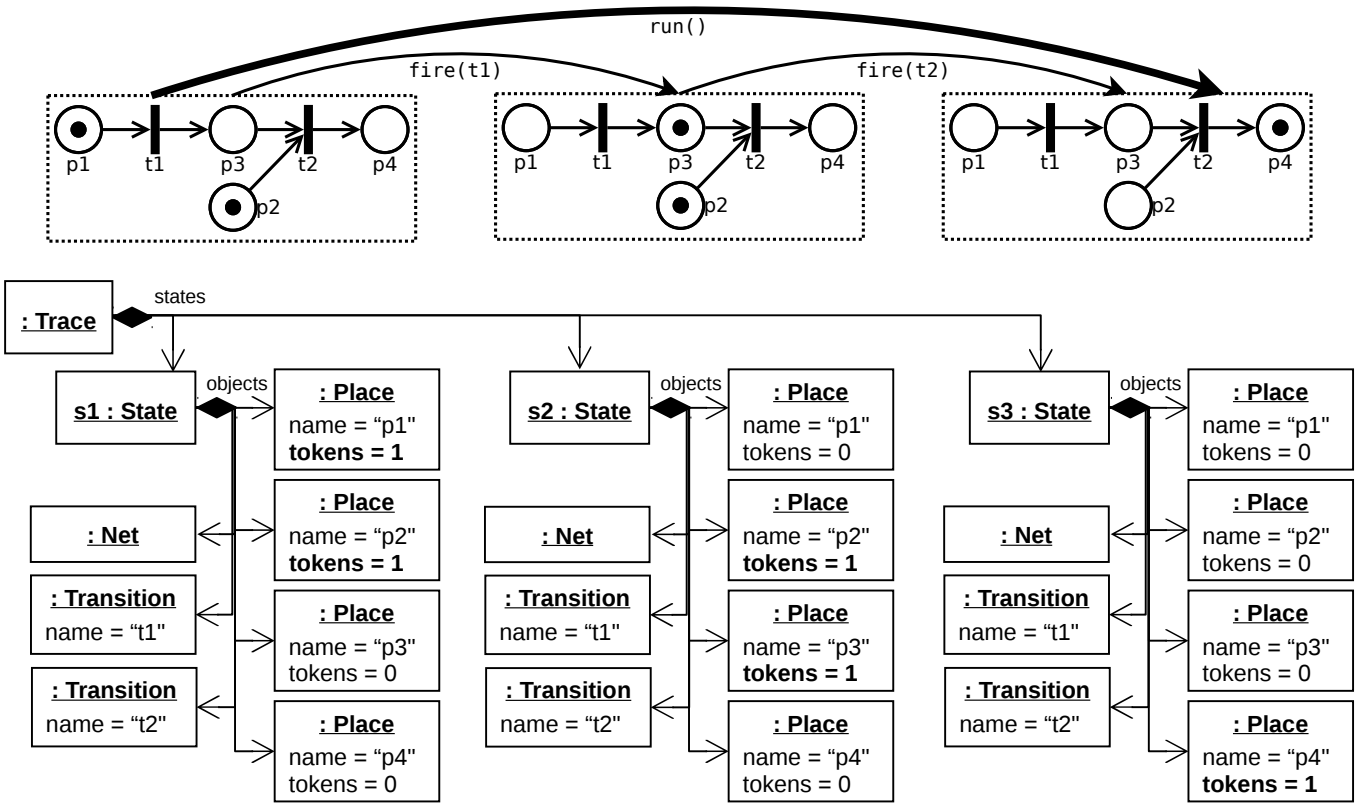
# Generic trace management

**State of the art**: generic trace metamodel [Langer et al.'14]



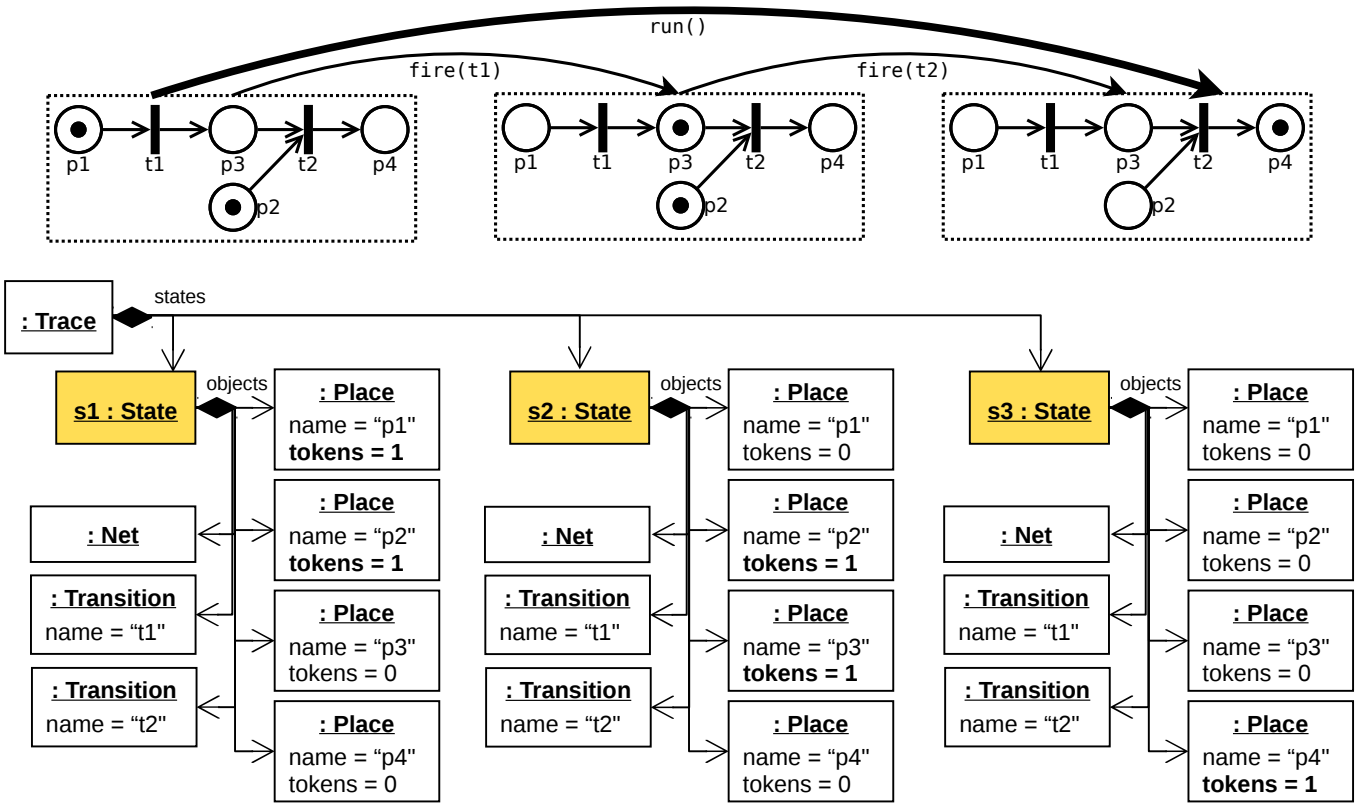

- States are stored in an unique **sequence**
- Each state is a **snapshot** (or clone) of the executed model

# Example of Petri net generic trace



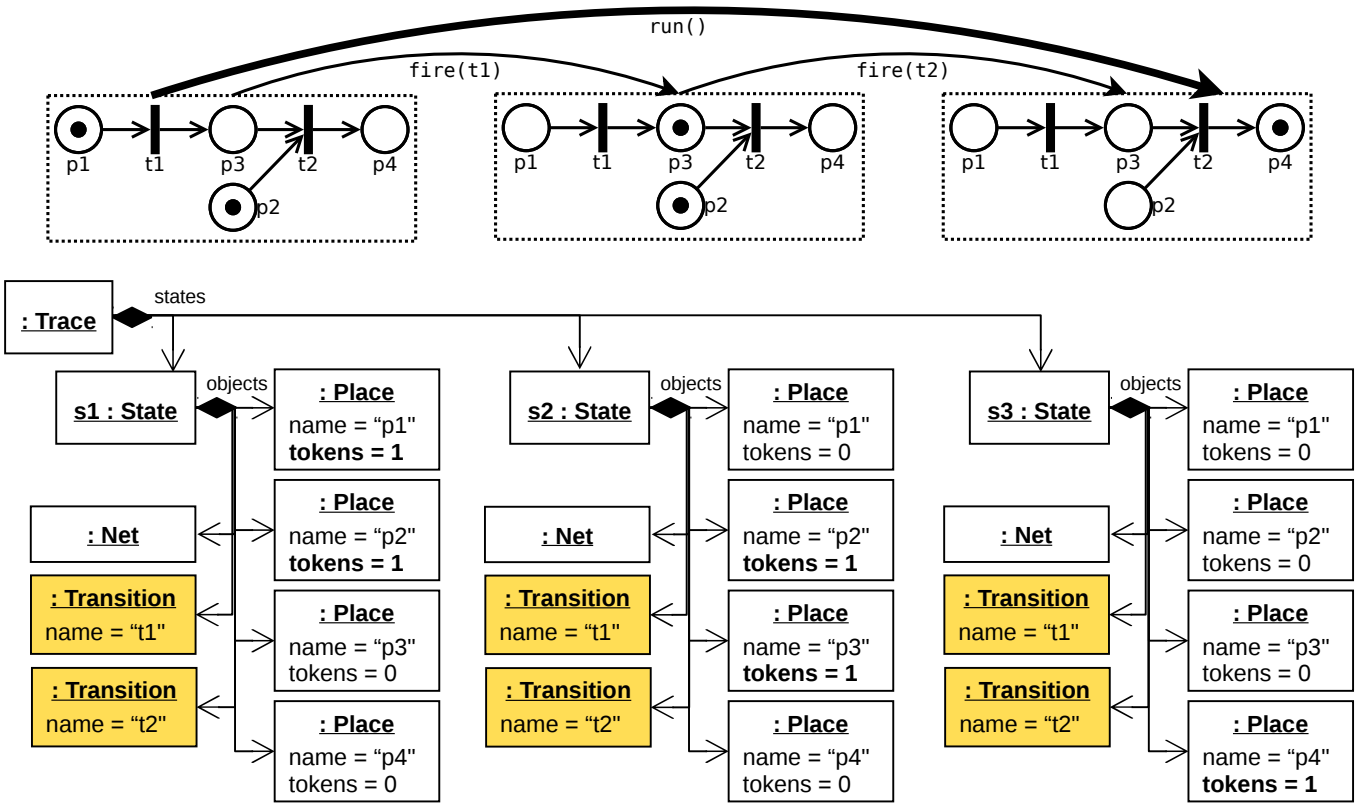

- **Scalability in time problem**: need to visit all states
- **Memory problem**: redundancy with both immutable and mutable data
- **Usability problem**: type checks and casting

# Example of Petri net generic trace



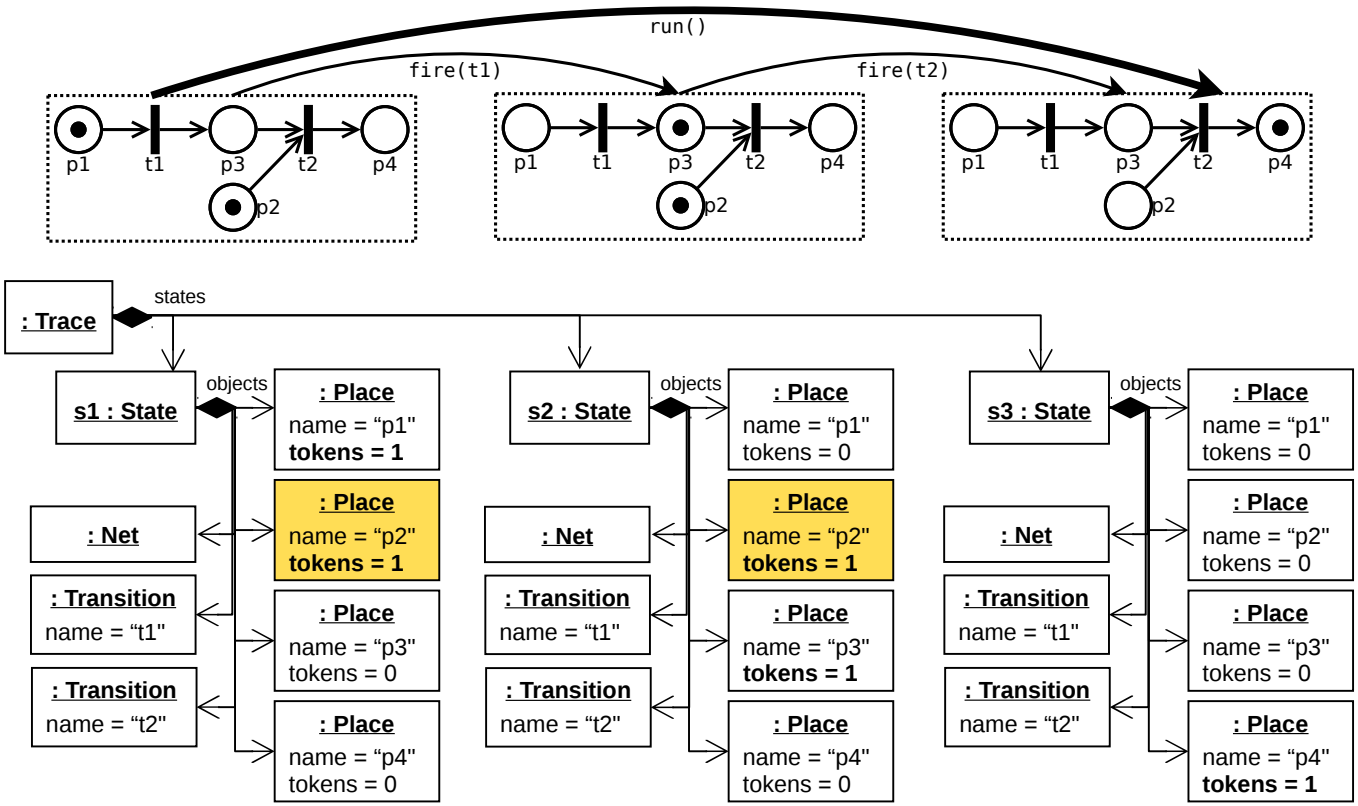

- **Scalability in time problem**: need to visit all states
- **Memory problem**: redundancy with both immutable and mutable data
- **Usability problem**: type checks and casting

# Example of Petri net generic trace



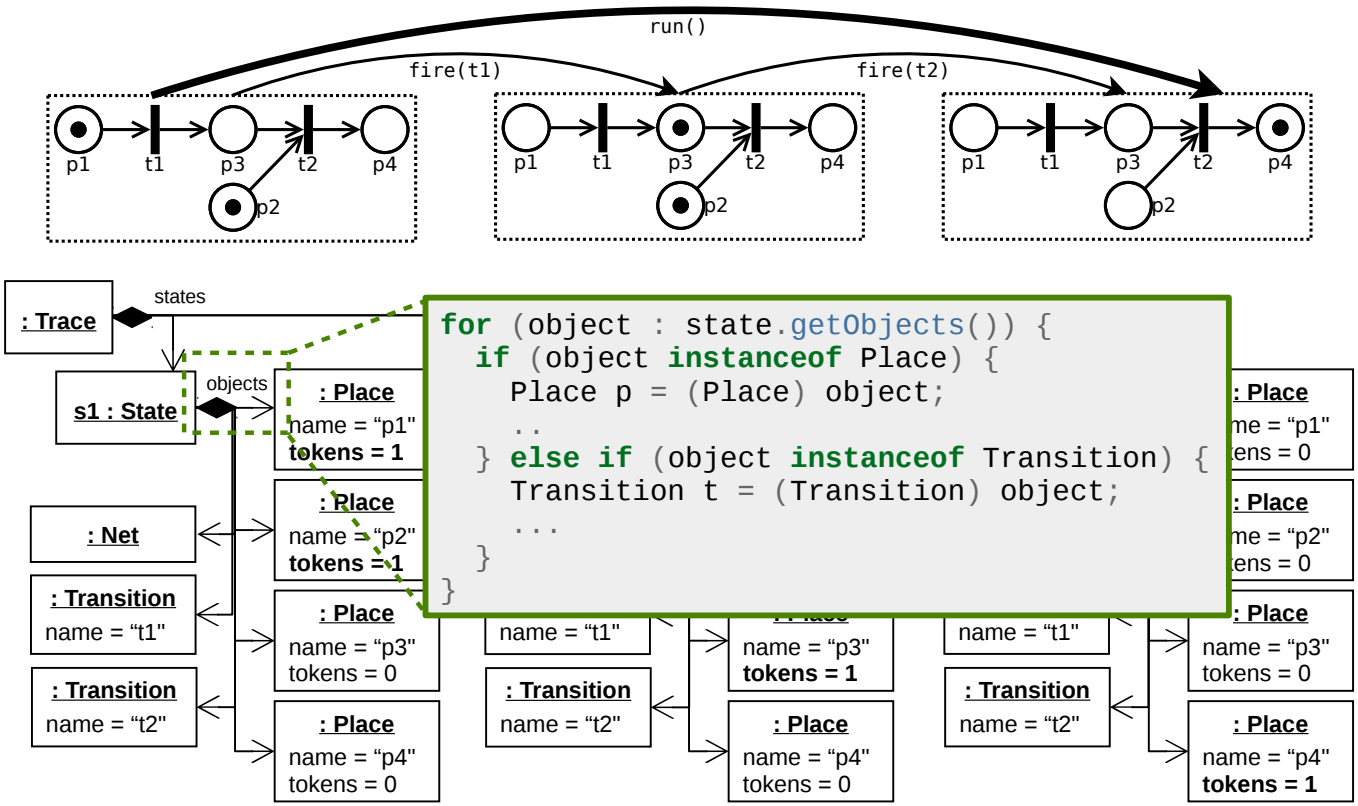

- **Scalability in time problem**: need to visit all states
- **Memory problem**: redundancy with both immutable and mutable data
- **Usability problem**: type checks and casting

# Example of Petri net generic trace



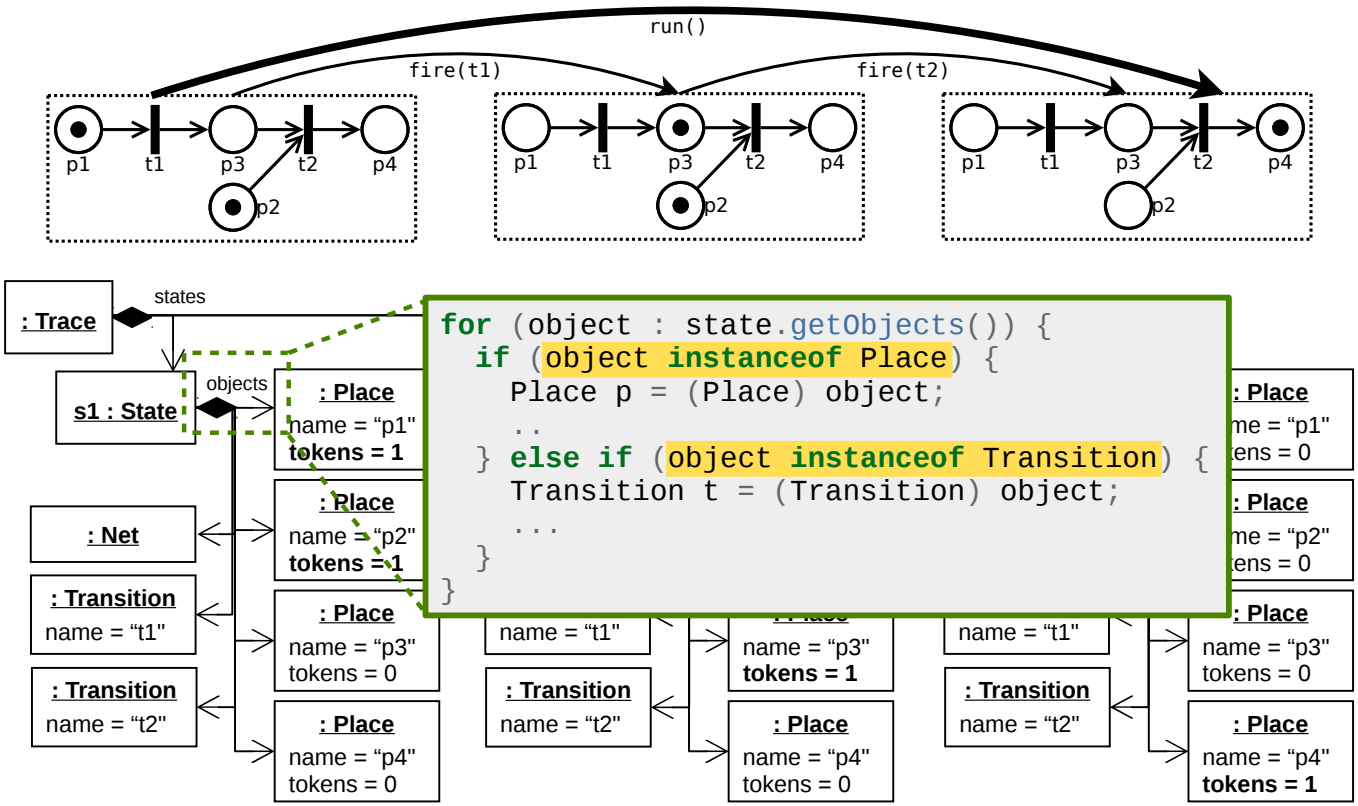

- **Scalability in time problem**: need to visit all states
- **Memory problem**: redundancy with both immutable and mutable data
- **Usability problem**: type checks and casting

# Example of Petri net generic trace



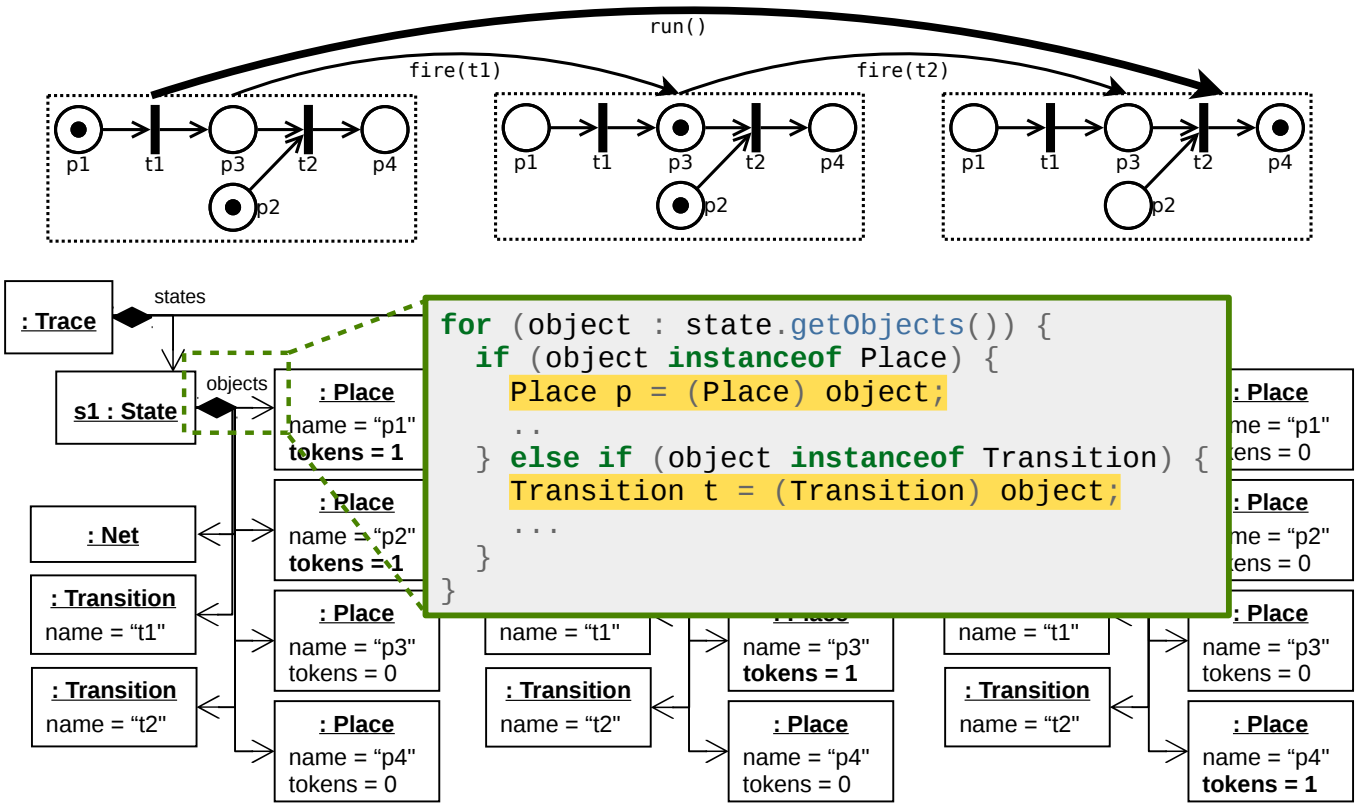

- **Scalability in time problem**: need to visit all states
- **Memory problem**: redundancy with both immutable and mutable data
- **Usability problem**: type checks and casting

# Example of Petri net generic trace




- **Scalability in time problem**: need to visit all states
- **Memory problem**: redundancy with both immutable and mutable data
- **Usability problem**: type checks and casting

# Example of Petri net generic trace




- **Scalability in time problem**: need to visit all states
- **Memory problem**: redundancy with both immutable and mutable data
- **Usability problem**: type checks and casting

# Approach: generating a domain-specific trace metamodel

## To provide usability and scalability in memory

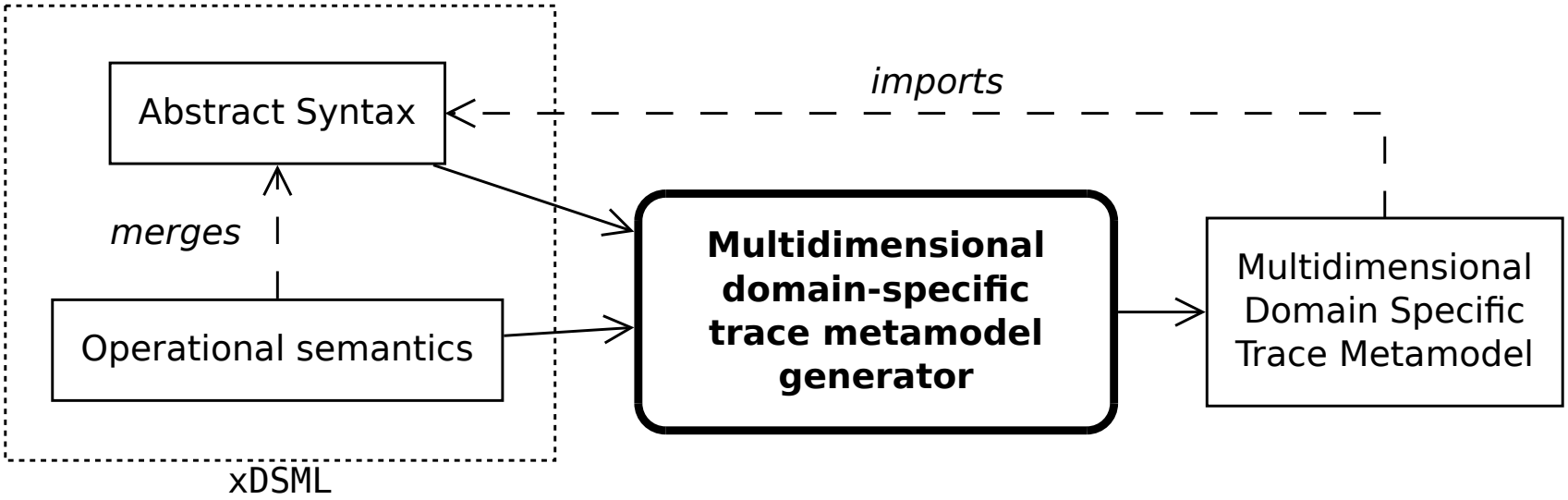
Generative approach to automatically derive a **domain-specific trace metamodel** for a given xDSML

- *Domain-specific*: domain concepts are directly accessible
- *Precise*: only concepts related to execution are kept
- *Automated*: save language engineers the design of a complex metamodel, which is time-consuming and error-prone

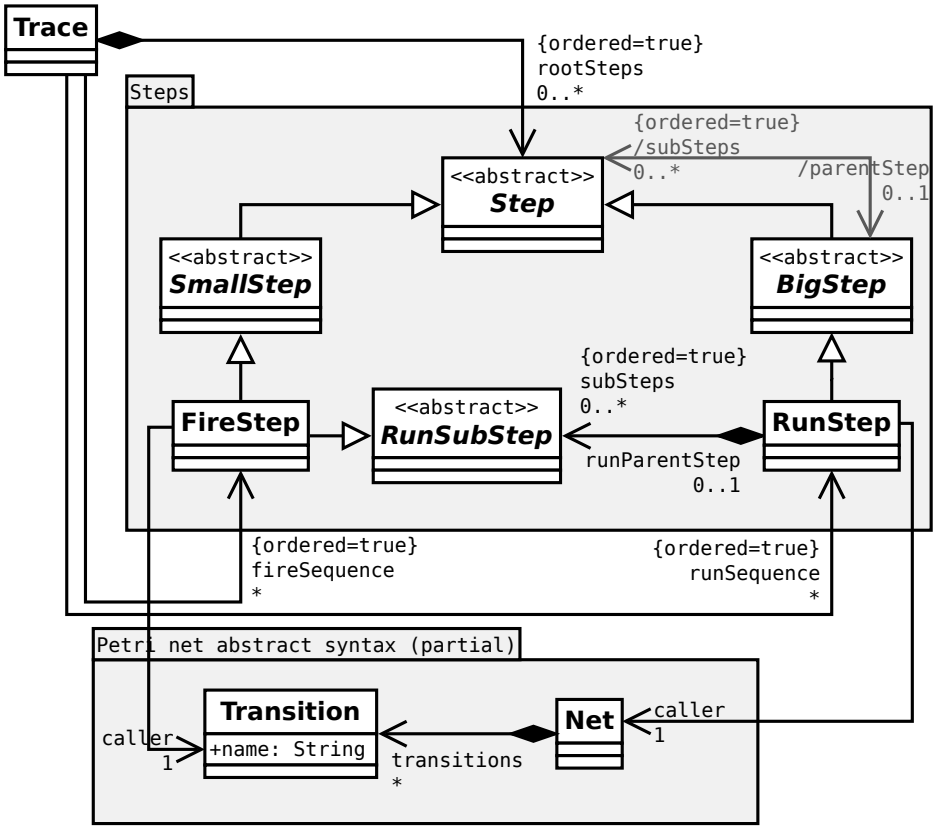
## To provide scalability in time

**Multidimensional** navigation facilities, e.g. browsing a trace according to the values reached by a specific mutable element

# Overview

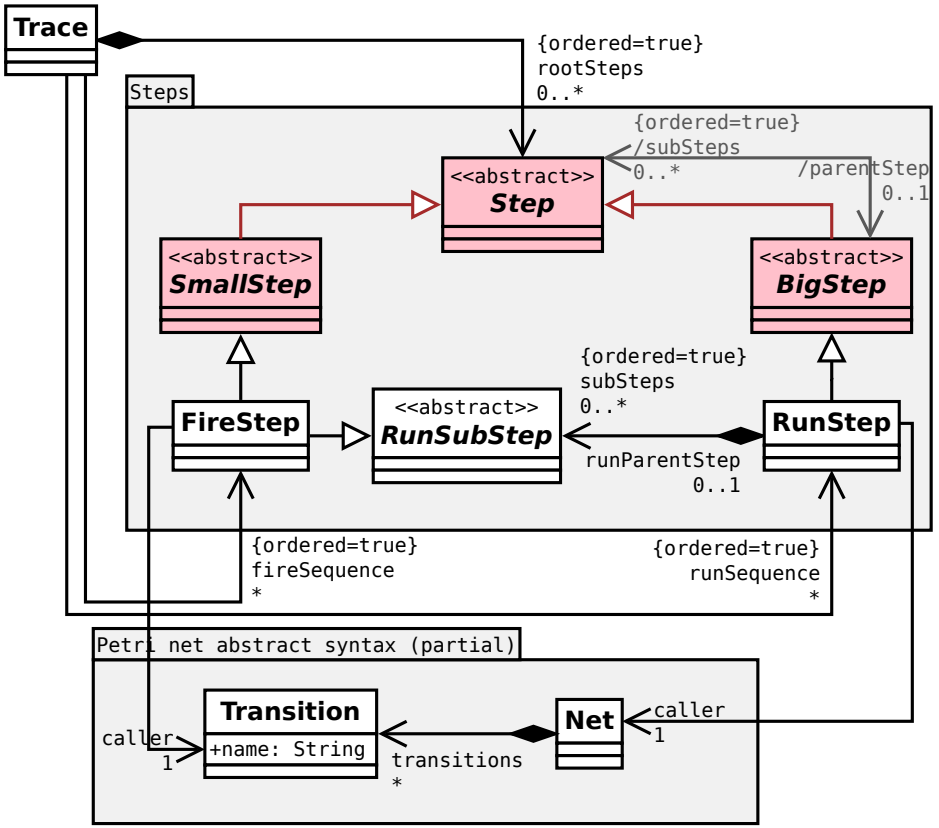

Abstract Syntax
imports
merges
Operational semantics
Multidimensional domain-specific trace metamodel generator
Multidimensional Domain Specific Trace Metamodel
xDSML

# Trace metamodel generation – Steps concepts



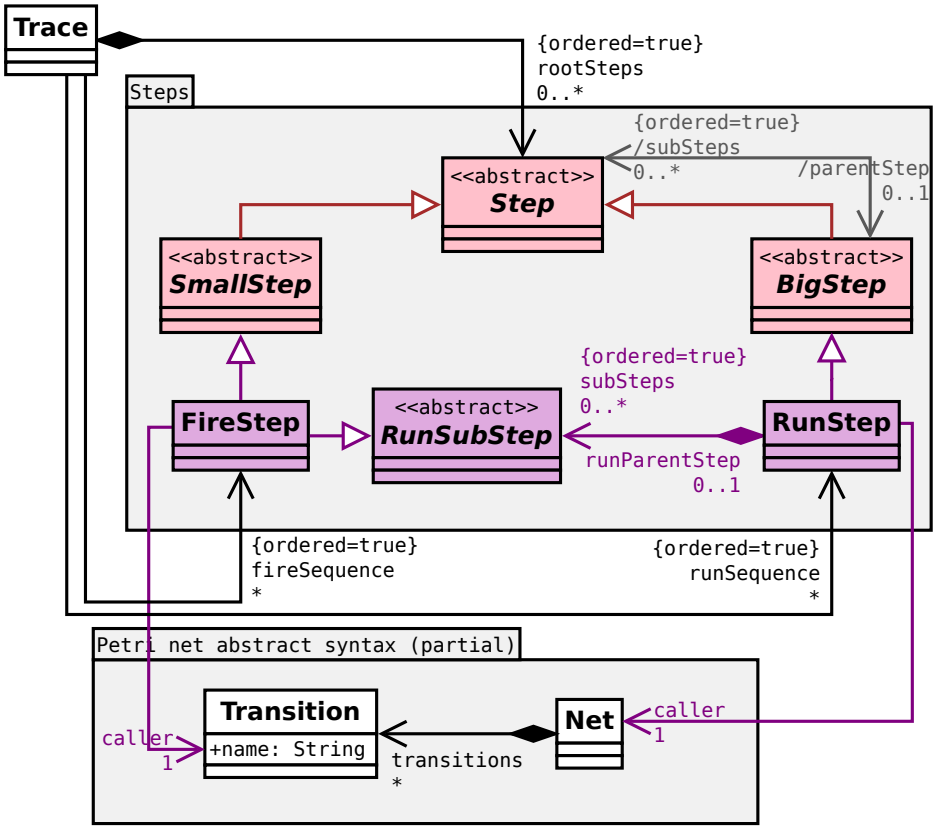

1. Base classes
   - **small step** = standalone transformation rule
   - **big step** = rule relying on other rules
2. Reification of rules into step classes
3. Steps made accessible as sequences or as a containment tree

# Trace metamodel generation – Steps concepts



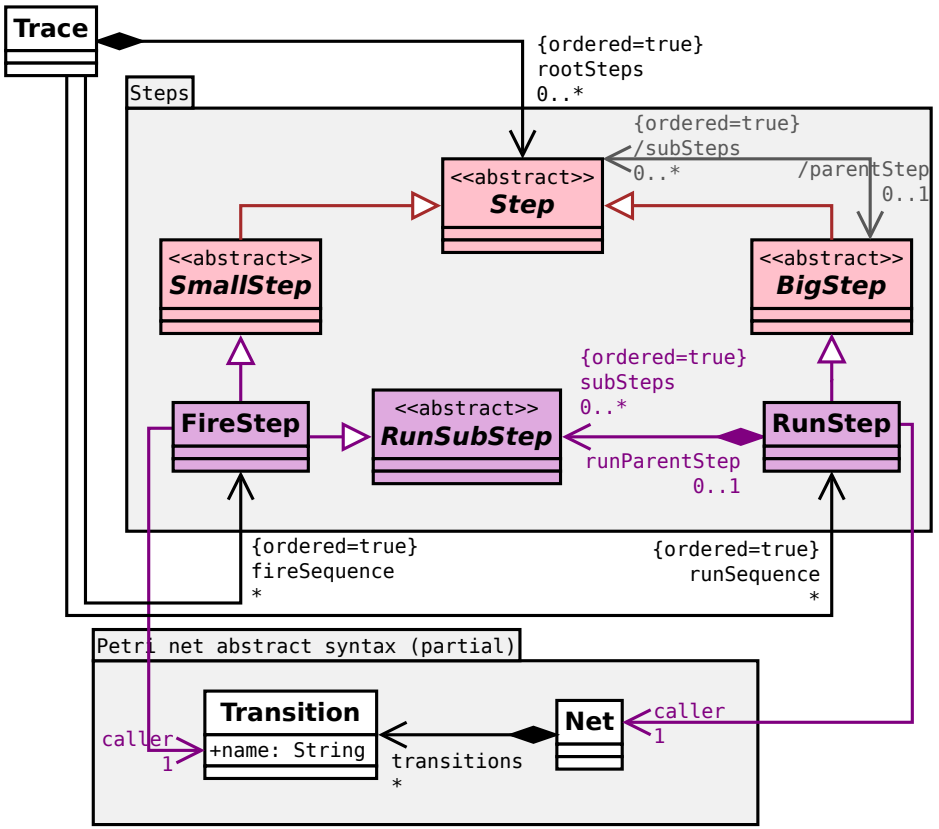

1. **Base classes**
   - **small step** = standalone transformation rule
   - **big step** = rule relying on other rules
2. Reification of rules into step classes
3. Steps made accessible as sequences or as a containment tree

# 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.

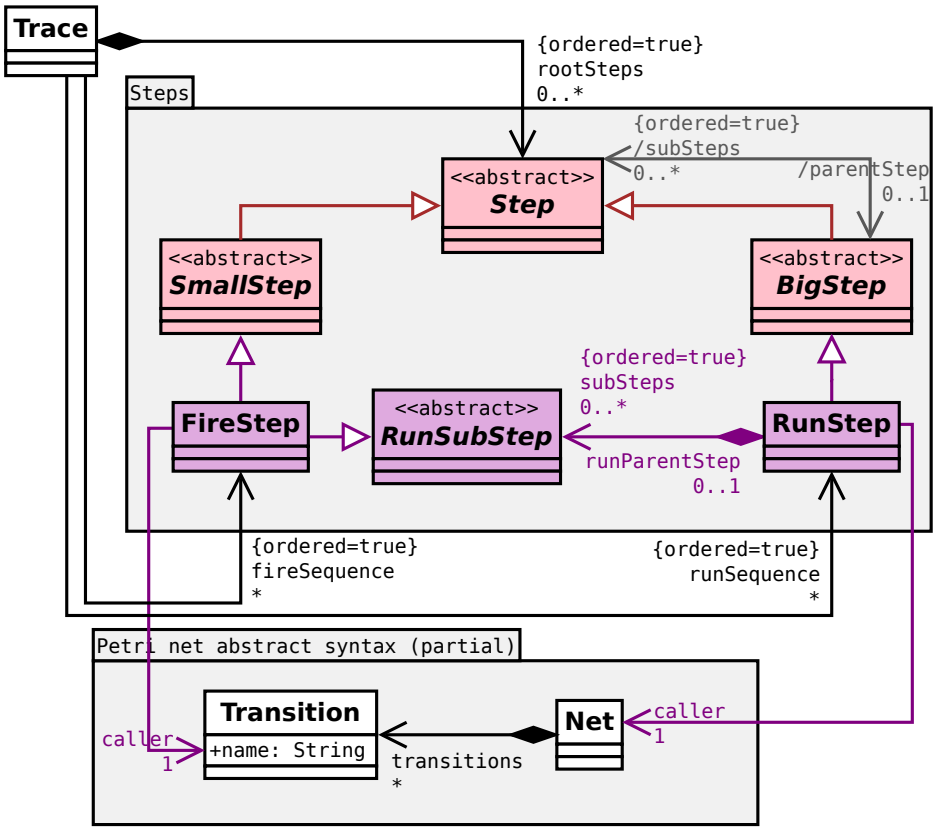
1. Analysis of the code

```
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




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

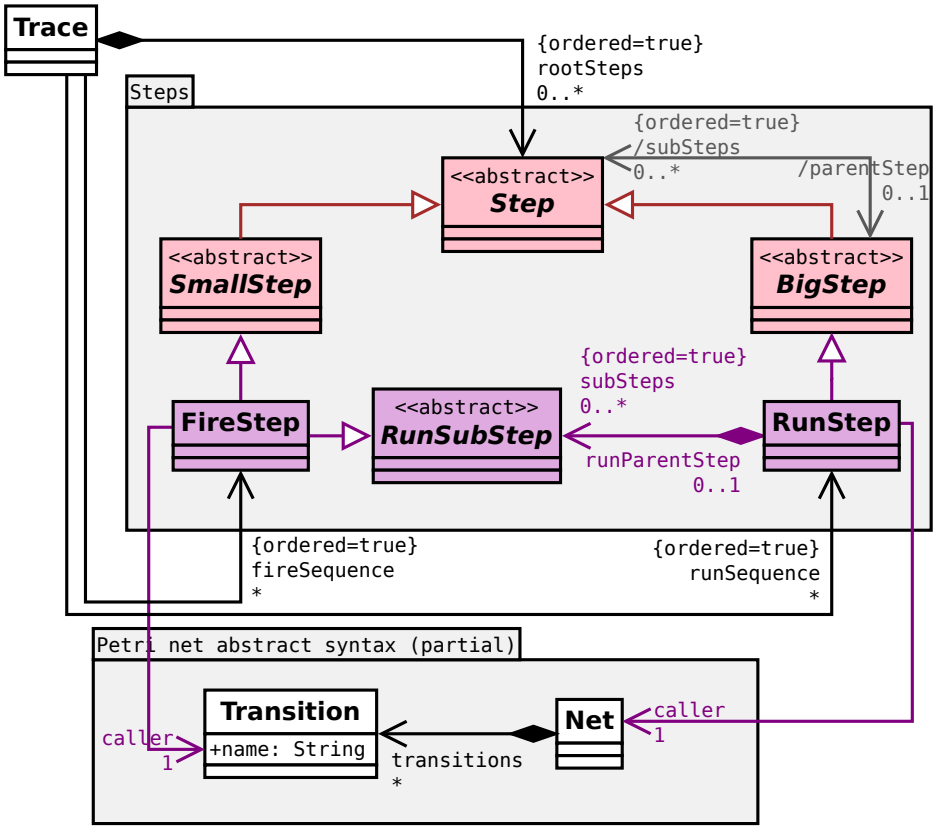
1. Analysis of the code

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

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

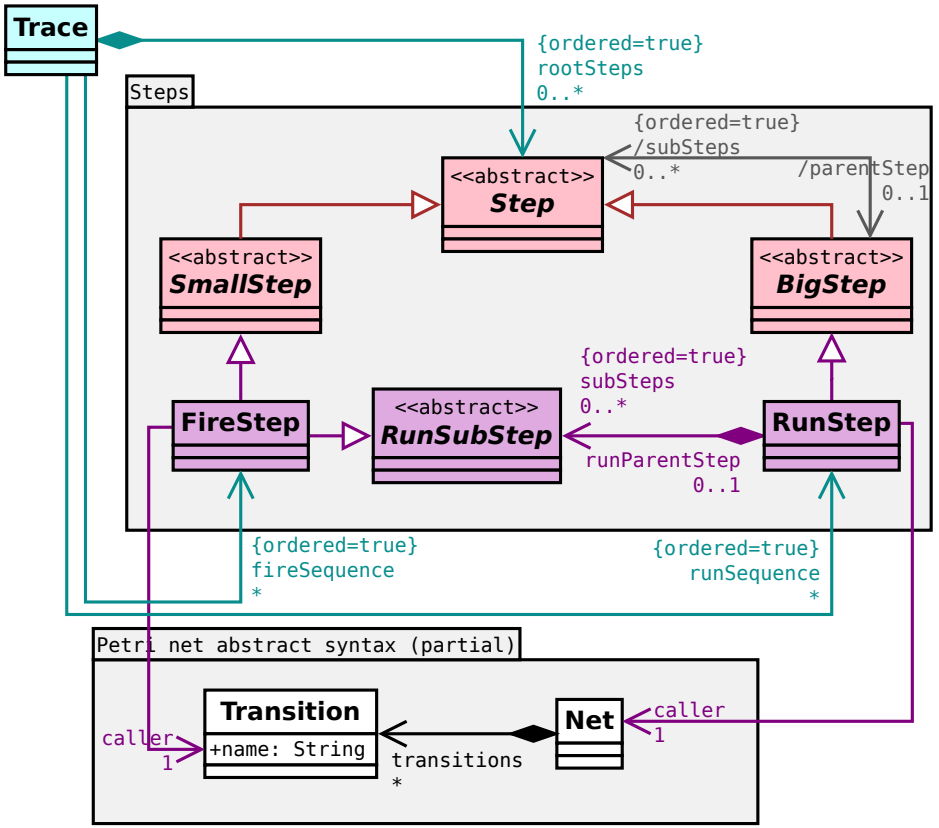
2. Creation of a **call graph**
3. **Pre-processing** of the call graph (e.g. methods overriding)
4. **Discovery** of big/small steps

# Trace metamodel generation – Steps concepts



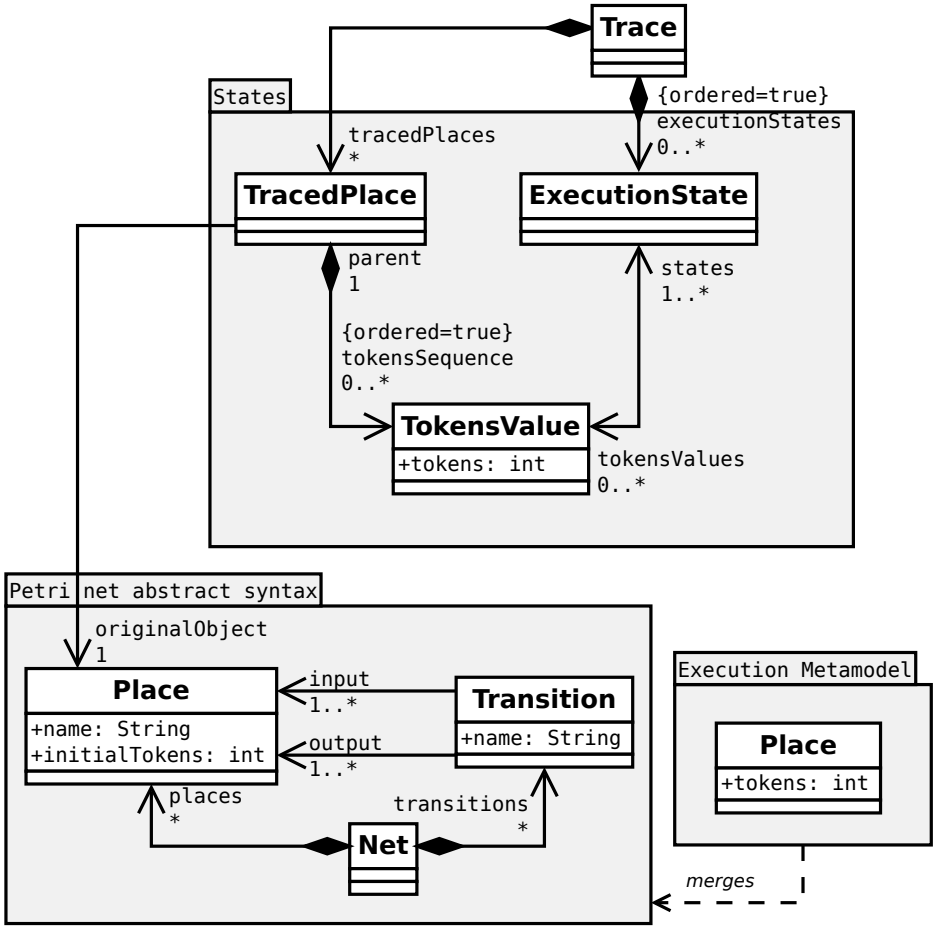

1. Base classes
   - **small step** = standalone transformation rule
   - **big step** = rule relying on other rules
2. Reification of rules into step classes
3. Steps made accessible as sequences or as a containment tree

# Trace metamodel generation – Steps concepts



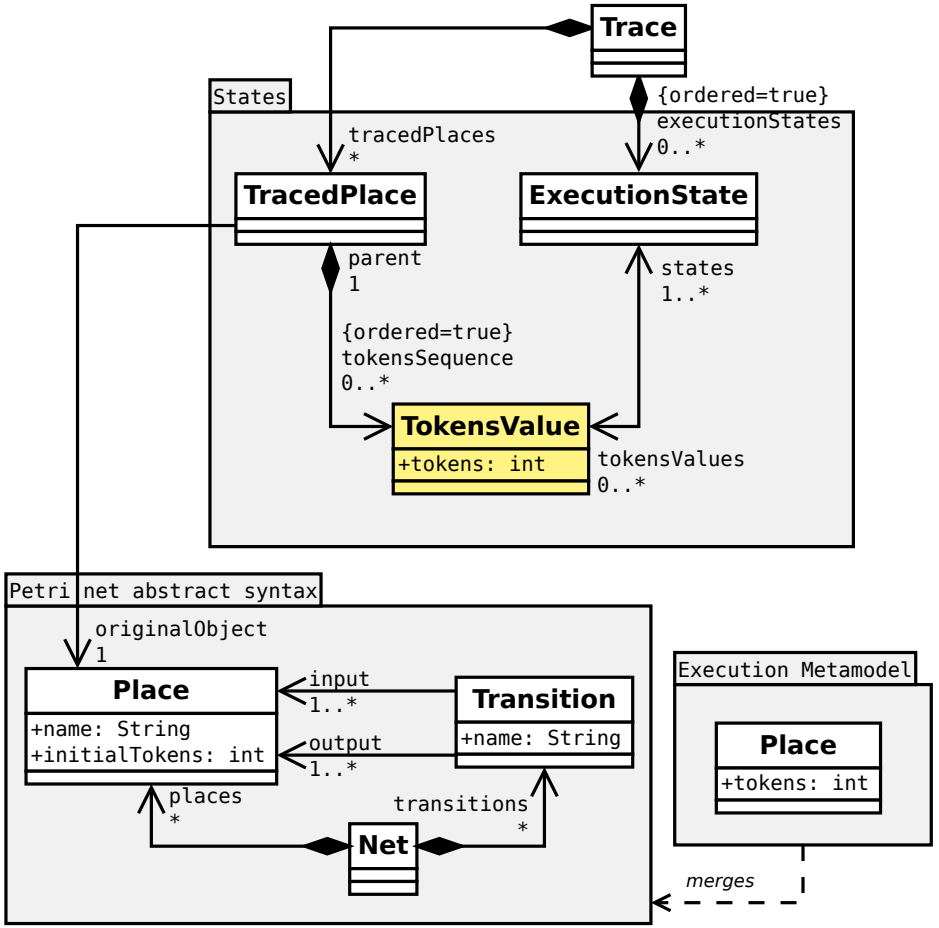

1. Base classes
   - **small step** = standalone transformation rule
   - **big step** = rule relying on other rules
2. Reification of rules into step classes
3. Steps made accessible as sequences or as a containment tree

# Trace metamodel generation – States concepts



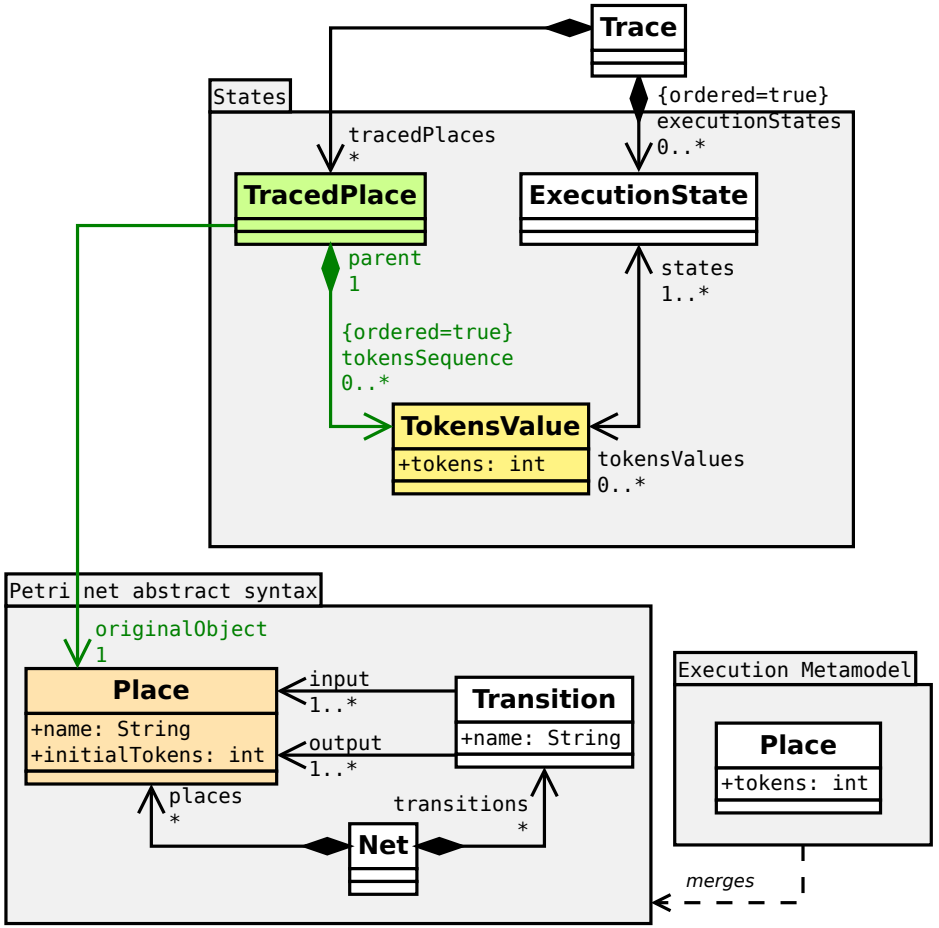

1. Reification of mutable properies into value classes
2. Values stored as sequences , for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

# Trace metamodel generation – States concepts



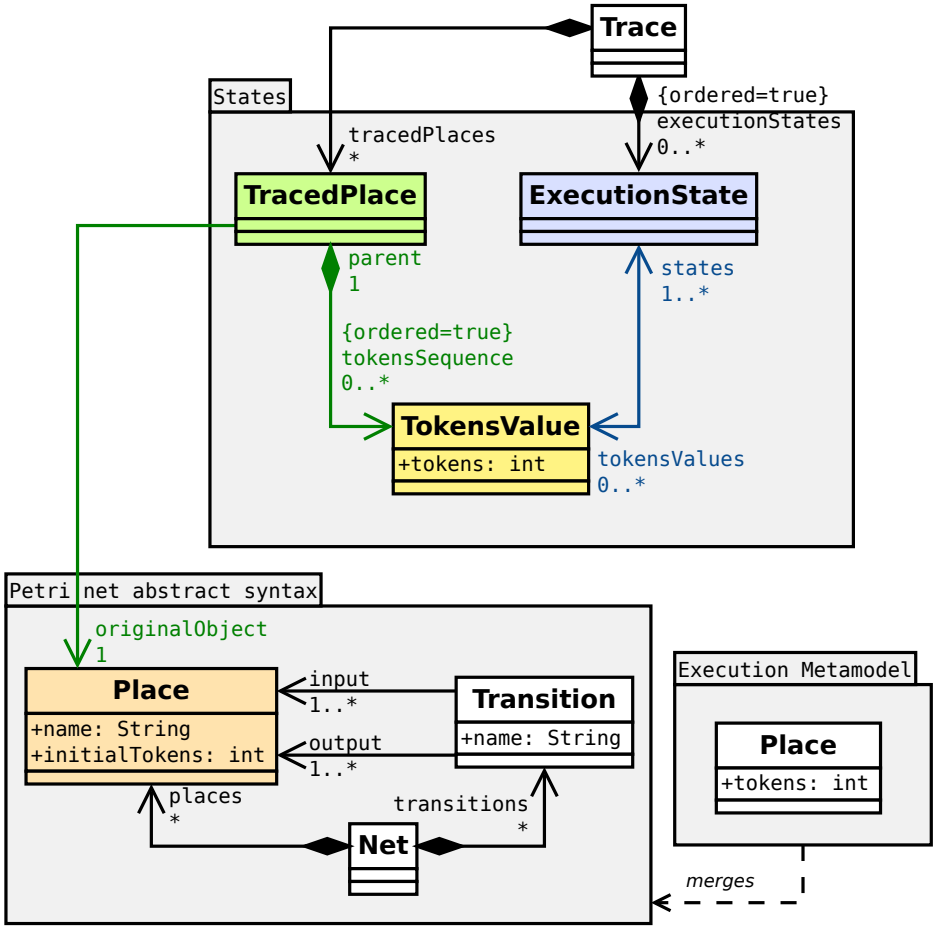

1. Reification of mutable properies into value classes
2. Values stored as sequences , for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

# Trace metamodel generation – States concepts



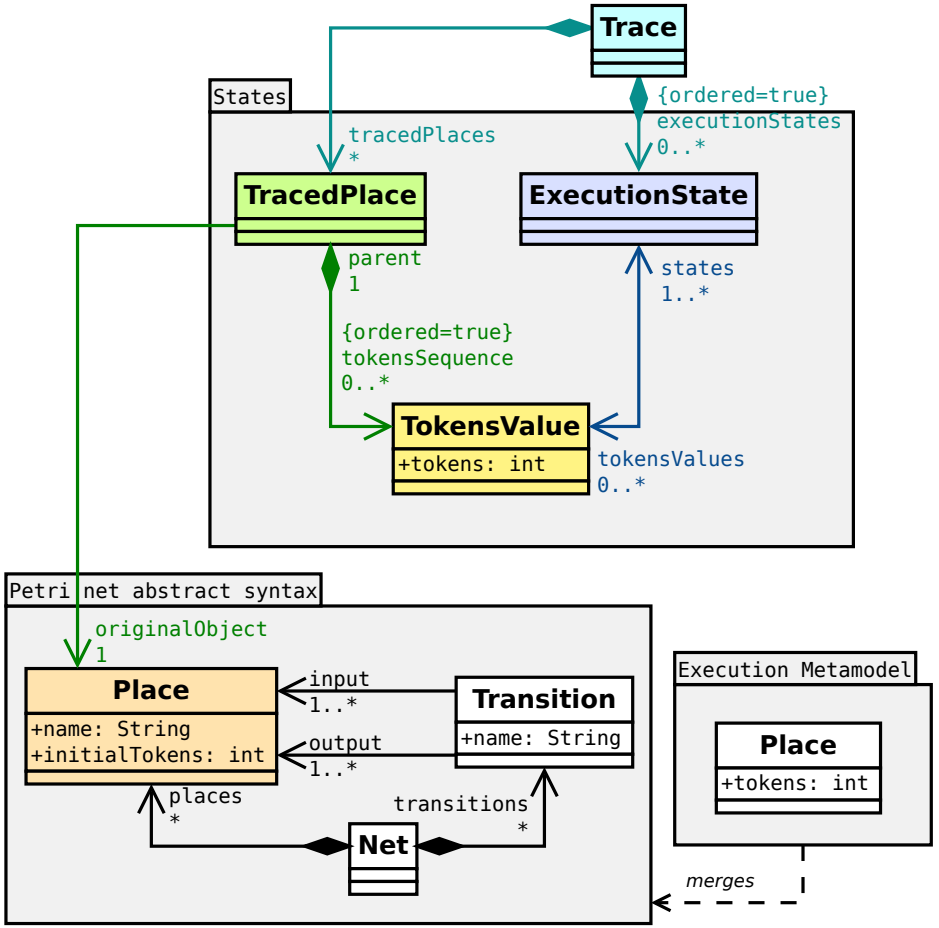

1. Reification of mutable properies into value classes
2. Values stored as sequences, for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

# Trace metamodel generation – States concepts



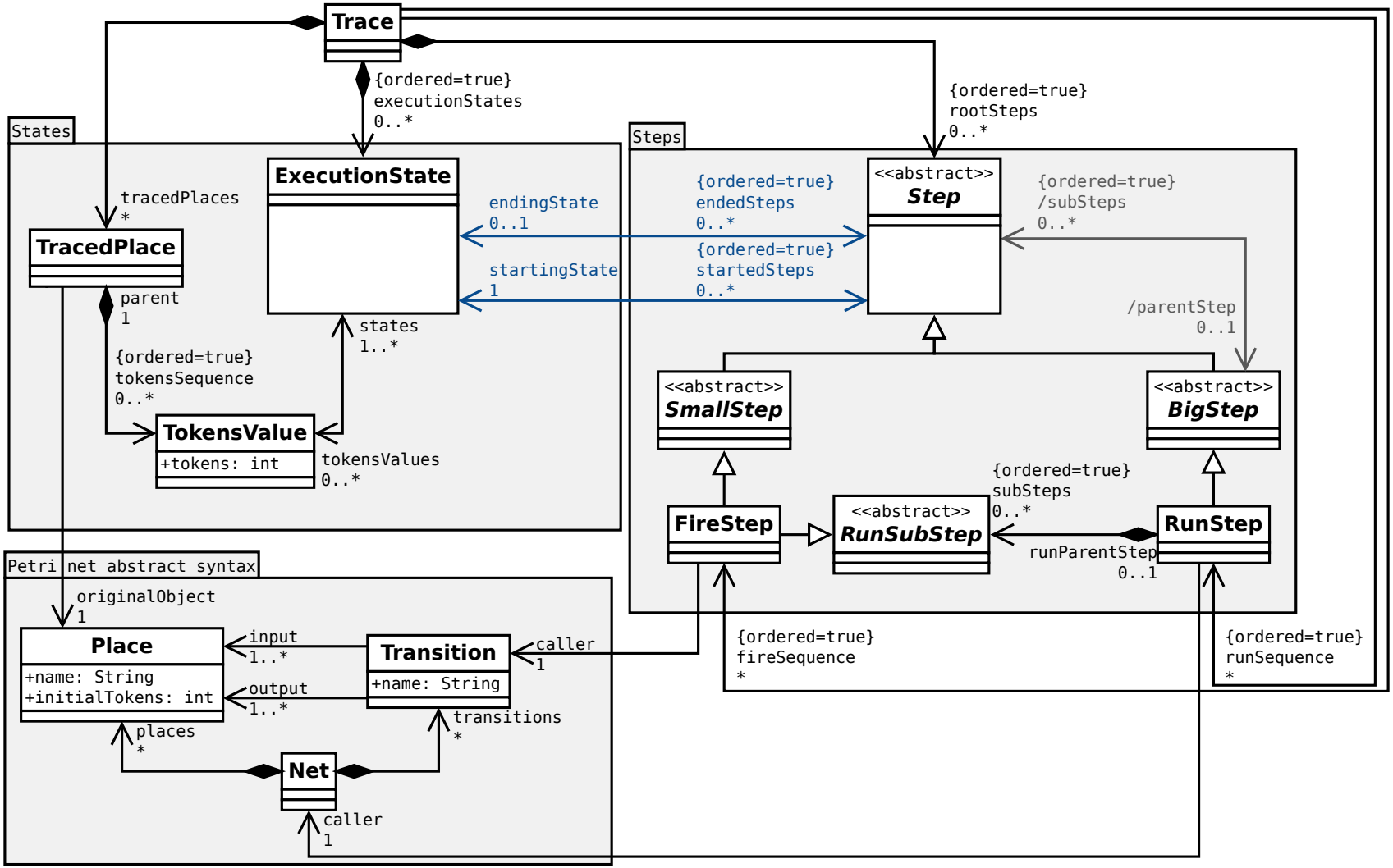

1. Reification of mutable properies into value classes
2. Values stored as sequences, for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

# Trace metamodel generation – States concepts

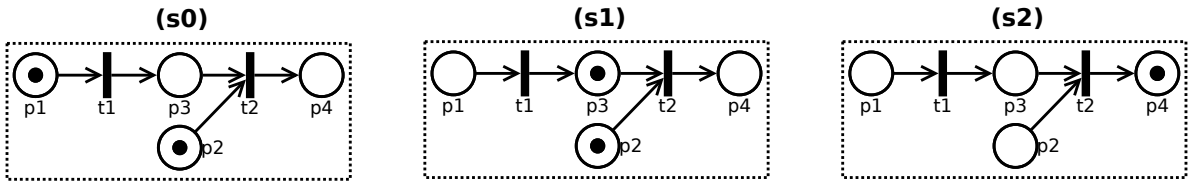


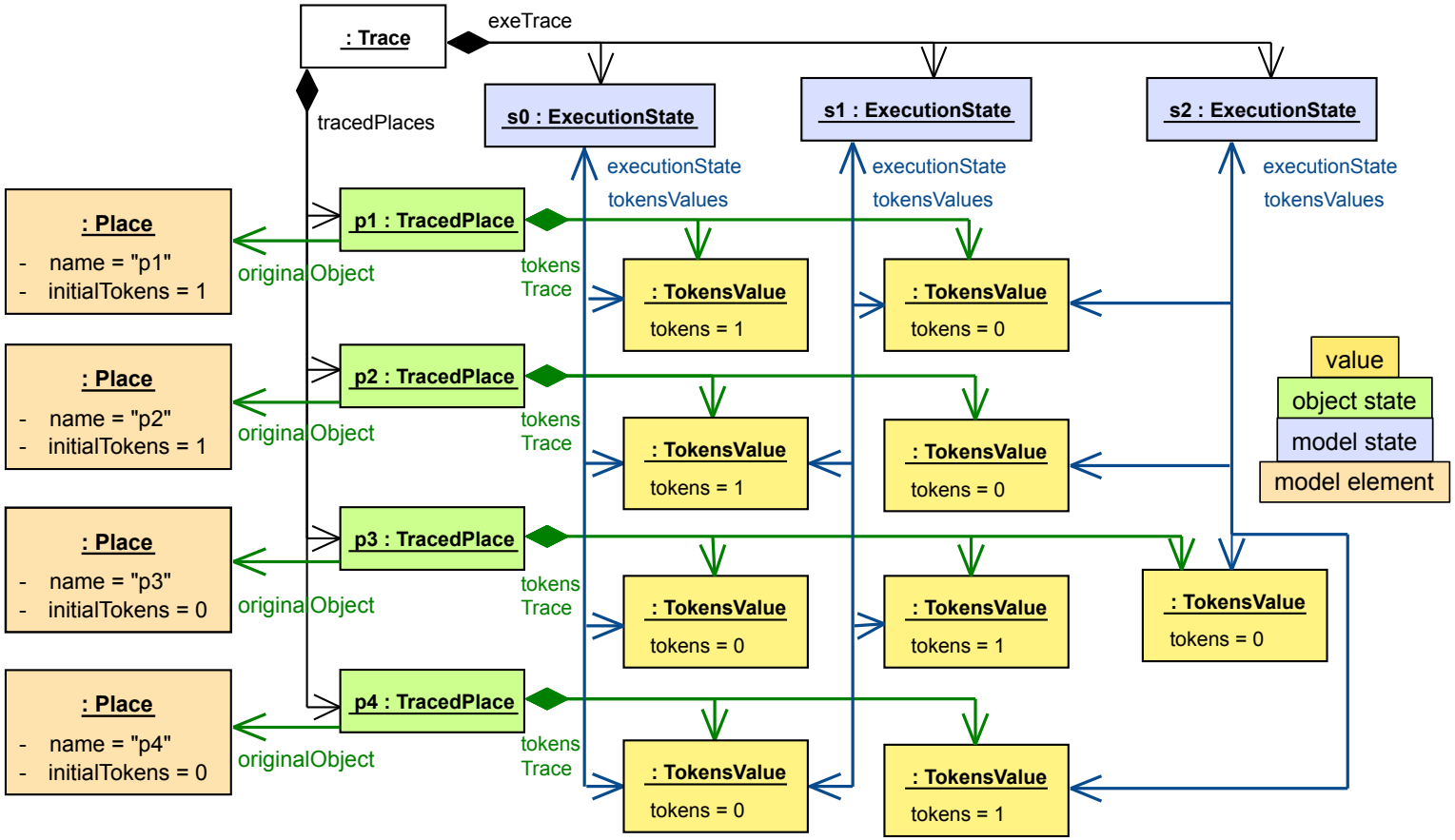

1. Reification of mutable properies into value classes
2. Values stored as sequences, for each model object
3. Execution state = set of values
4. Can be browsed by states or value sequences

# Resulting Petri net trace metamodel



# Example of Petri net trace (states only)



# Conclusion

## Summary

- **Precise capture** of the domain of the traces of an xDSML
- **Multidimensional navigation facilities** to ease queries

## Questions

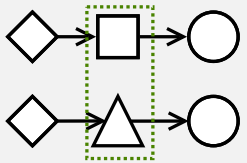
- **Usability** and **memory consumption** improved with domain-specificness?
- **Processing time** reduced thanks to multidimensional navigation?

$\Rightarrow$ Application to two different V&V techniques + measurements

# Outline



Enhanced semantic differencing



Advanced and efficient omniscient debugging [SLE'15]

# Evaluation

## Research questions (summarized)

As compared to a generic clone-based trace metamodel:

- **RQ1: Usability** of generated trace metamodels
- **RQ2: Scalability in time** of trace manipulations
- **RQ3: Scalability in memory** of traces

## Considered V&V techniques

1. **Semantic differencing**
   - Manually defined trace manipulations
   - Evaluation of RQ1 and RQ2
2. **Omniscient debugging**
   - Generated trace manipulations
   - Evaluation of RQ2 and RQ3

# Evaluation

## Research questions (summarized)

As compared to a generic clone-based trace metamodel:

- **RQ1: Usability** of generated trace metamodels
- **RQ2: Scalability in time** of trace manipulations
- **RQ3: Scalability in memory** of traces

## Considered V&V techniques

1. **Semantic differencing**
   - Manually defined trace manipulations
   - Evaluation of RQ1 and RQ2
2. **Omniscient debugging**
   - Generated trace manipulations
   - Evaluation of RQ2 and RQ3

# Semantic model differencing

## Context

- **Model differencing**: analyzing and understanding the changes made to a model during its development
- Mostly done syntactically (e.g. version control systems)

## The case of xDSMLs

- A change in an executable model $\equiv$ a change in its behavior
- Hence, model differencing must be done **semantically**
- One approach: **traces comparison** before and after a change.

# Semantic model differencing

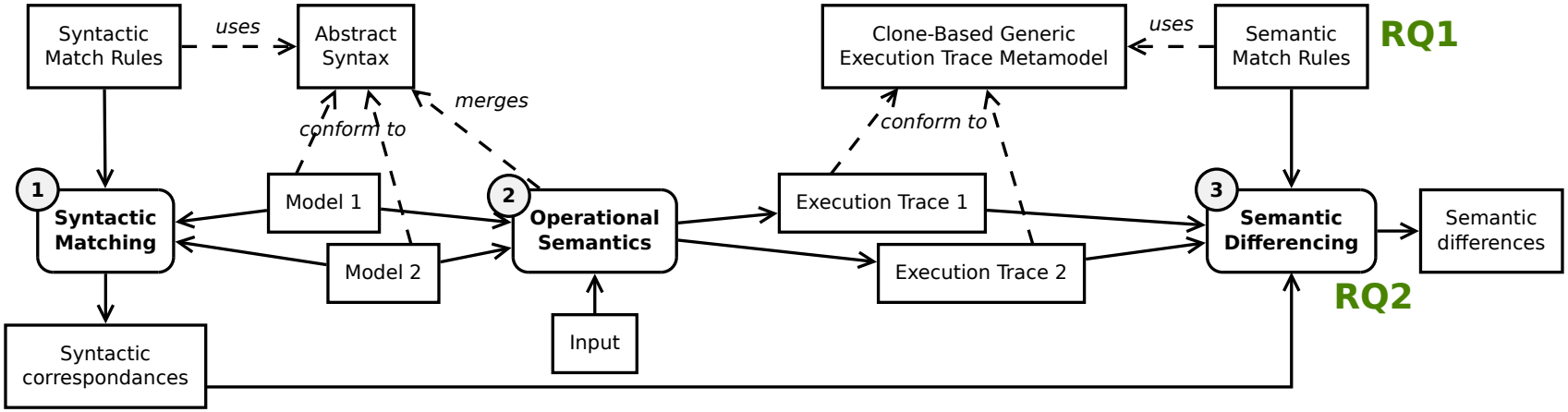
## Context

- **Model differencing**: analyzing and understanding the changes made to a model during its development
- Mostly done syntactically (e.g. version control systems)

## The case of xDSMLs

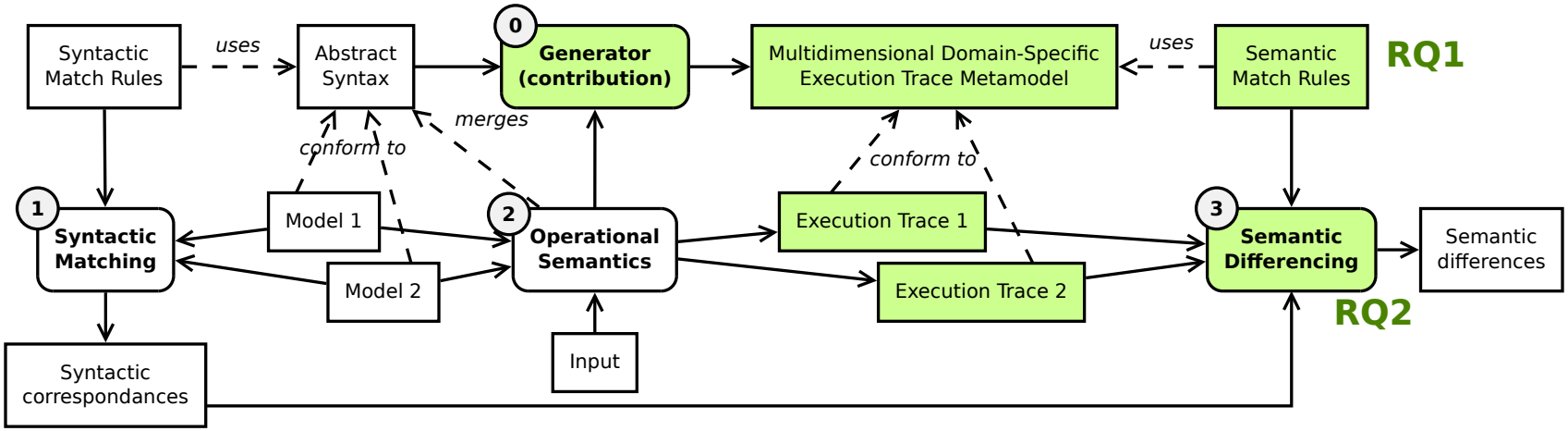
- A change in an executable model $\equiv$ a change in its behavior
- Hence, model differencing must be done **semantically**
- One approach: **traces comparison** before and after a change.

# Existing base approach [Langer et al.'14]




- Requires the **manual definition of semantic match rules** specific to the xDSML
- Relies on a **generic trace metamodel**
  - *usability (RQ1)*: match rules are hard to write
  - *scalability in time (RQ2)*: match rules are long to execute

# Enhanced approach




## Enhancement

- **Additional prior step**: generation of a *domain-specific trace metamodel* with our contribution
- **New semantic match rules** relying on multiple dimensions
- What we measure:
  - *usability (RQ1)*: **complexity of the match rules**
  - *scalability in time (RQ2)*: **execution time of the match rules**

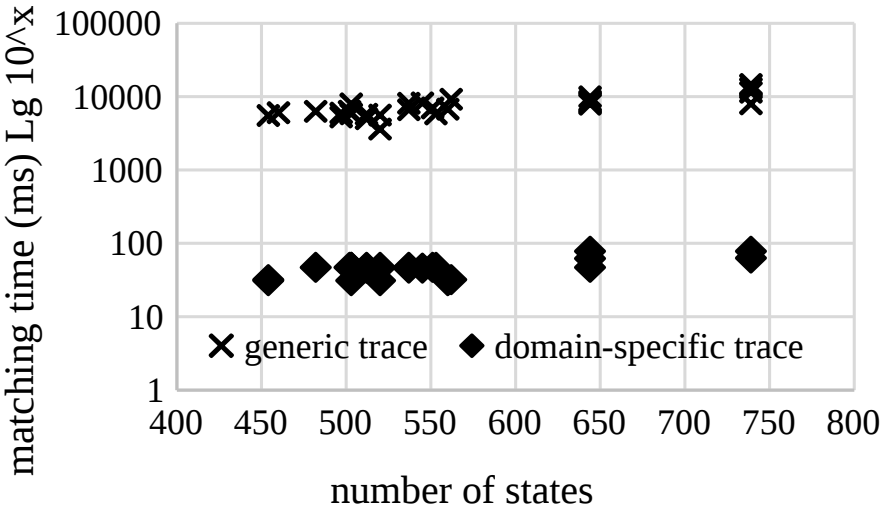
# Evaluation results

**Case study:** fUML xDSML and models from [Maoz et al.'11]

*Complexity of rules (RQ1)*

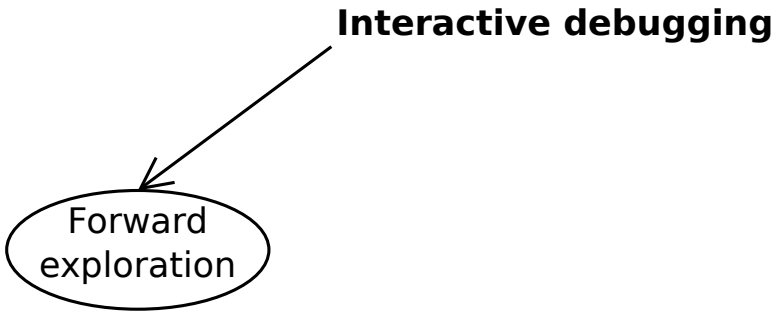
| Elements | Generic | Domain-specific |
|---|---|---|
| Lines of code | 90 | 44 |
| Statements | 35 | 16 |
| Operations | 8 | 3 |
| Operation calls | 35 | 24 |
| Loops | 5 | 4 |
| Type checks | 4 | 1 |
| Conditionals | 11 | 3 |

*Rules execution time (RQ2)*

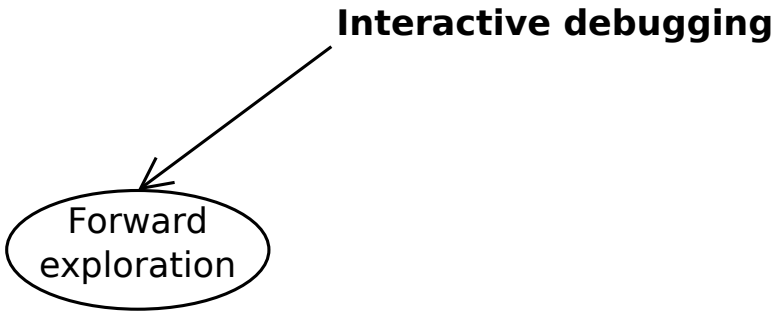


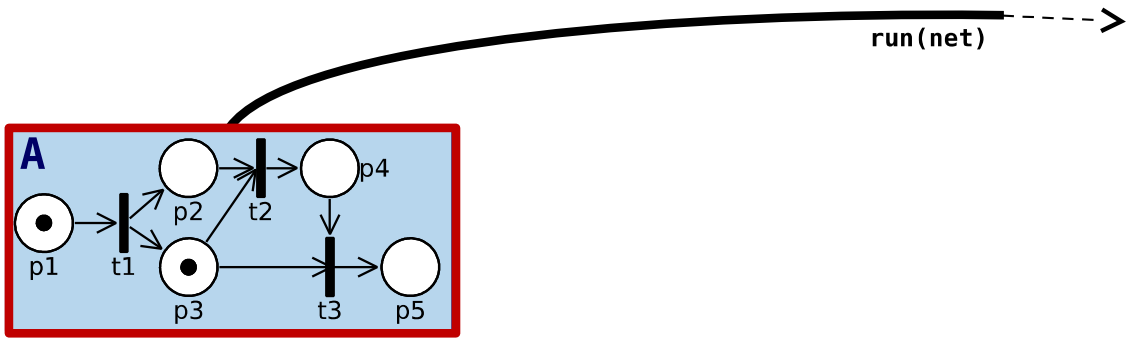
# Model omniscient debugging: terminology and scope

# Model omniscient debugging: terminology and scope

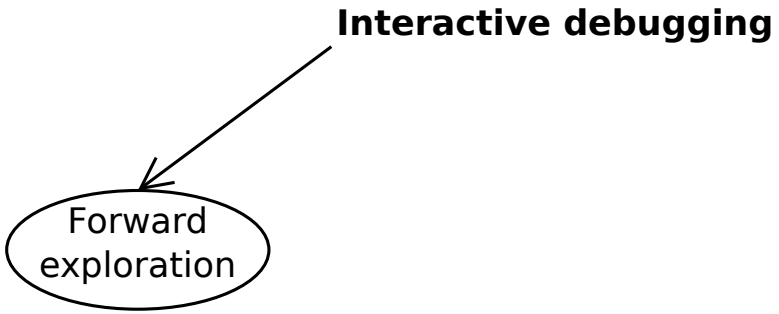


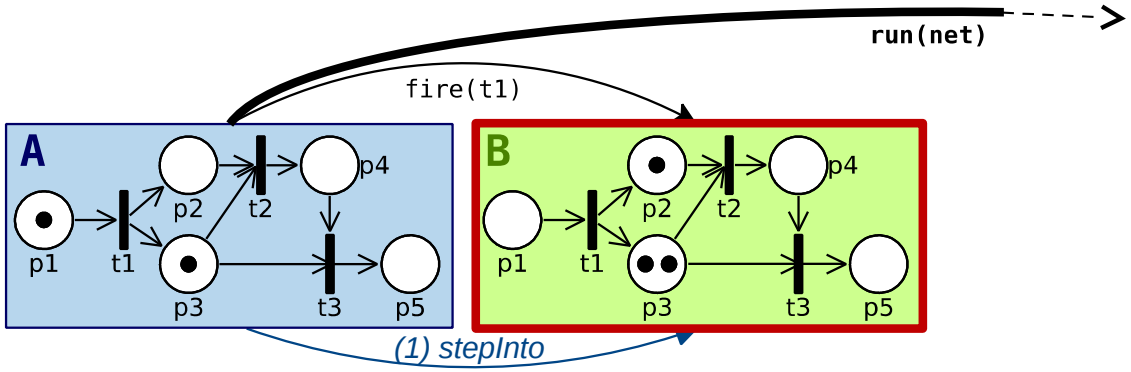
# Model omniscient debugging: terminology and scope



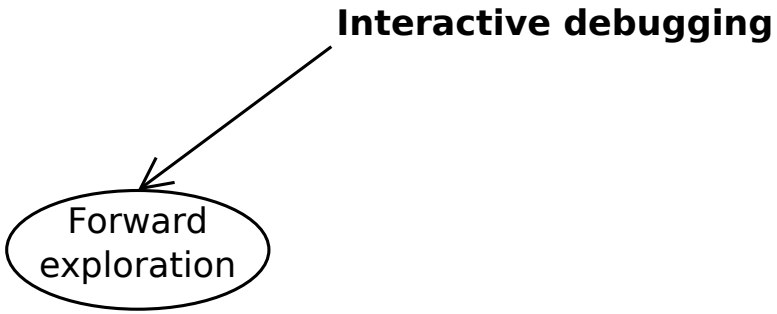



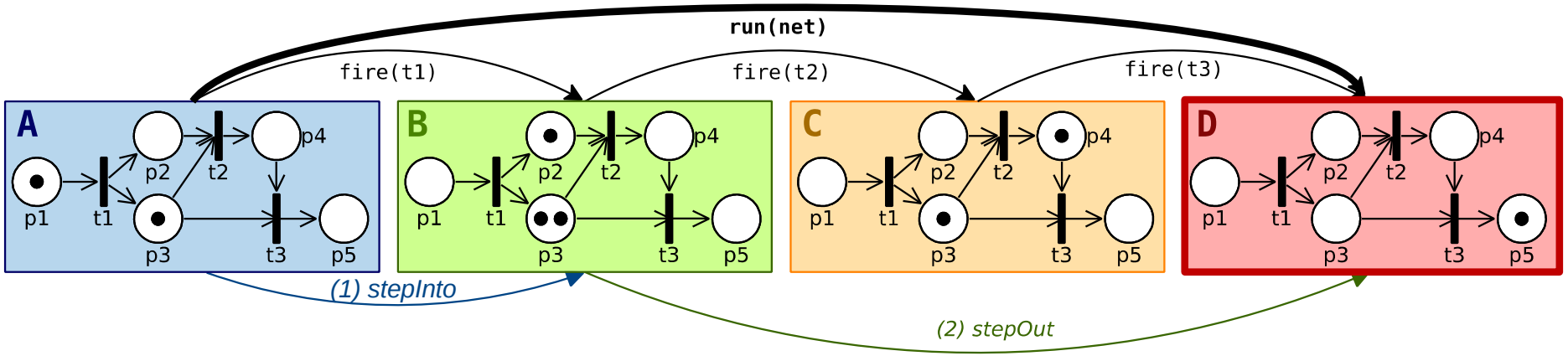
# Model omniscient debugging: terminology and scope



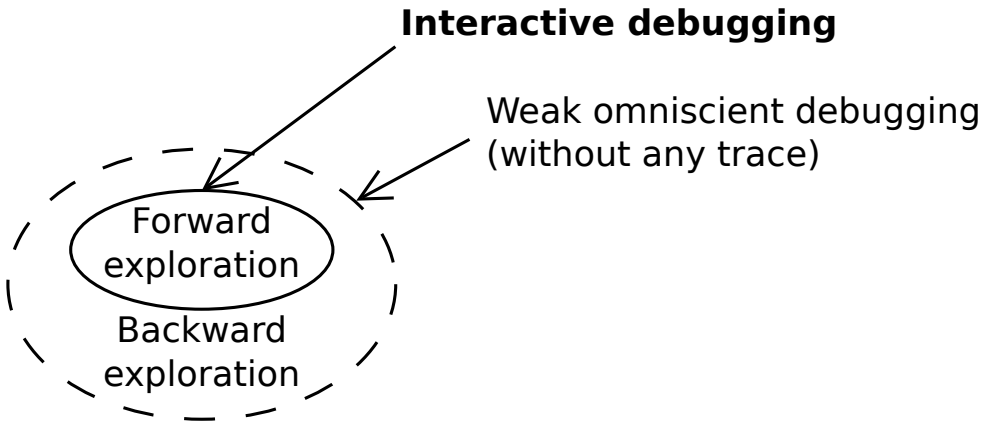



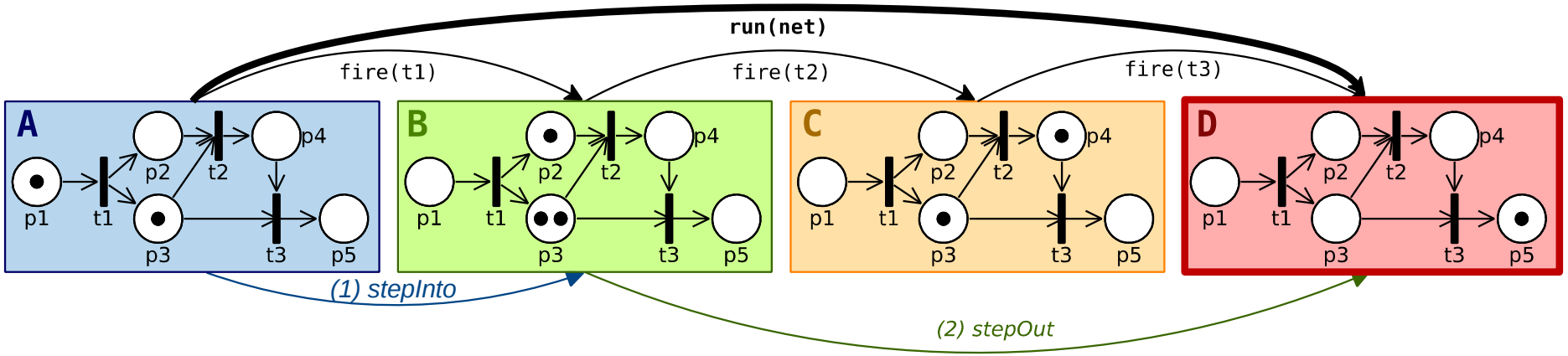
# Model omniscient debugging: terminology and scope



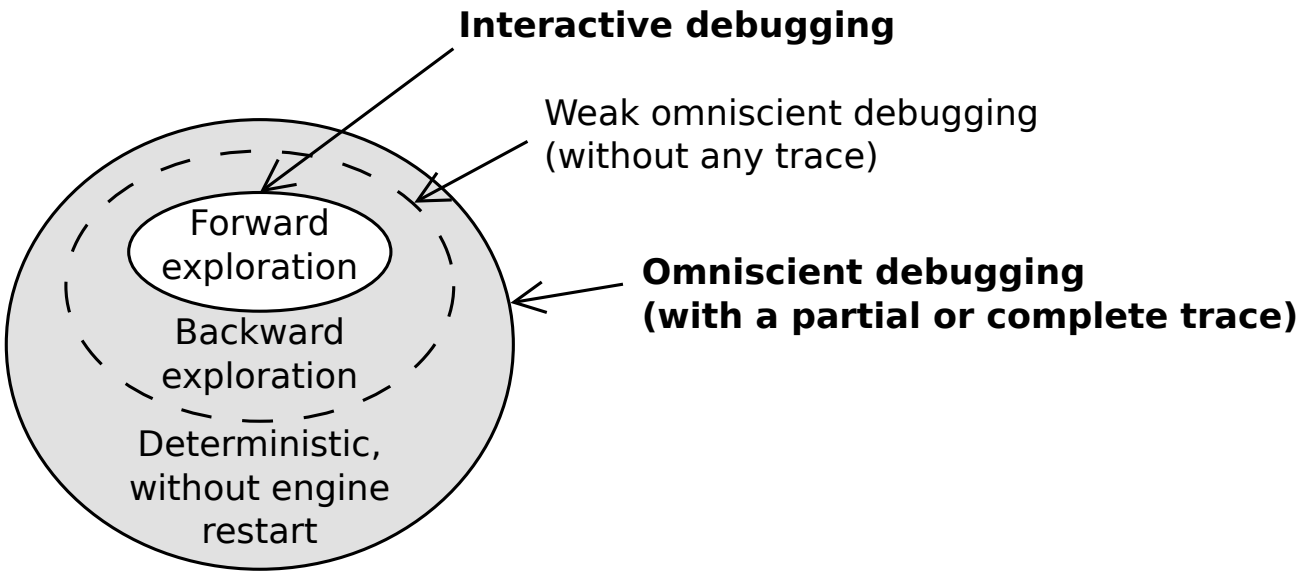



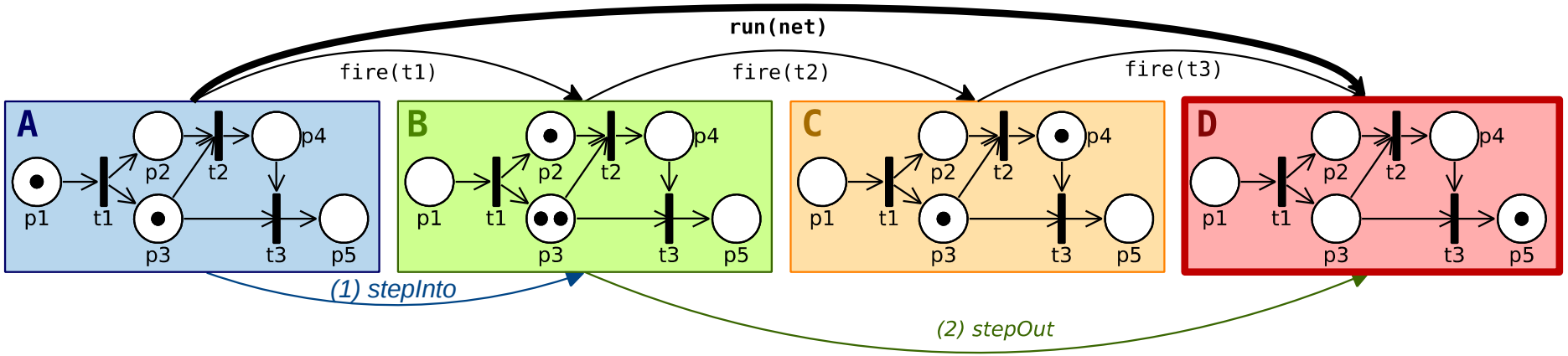
# Model omniscient debugging: terminology and scope



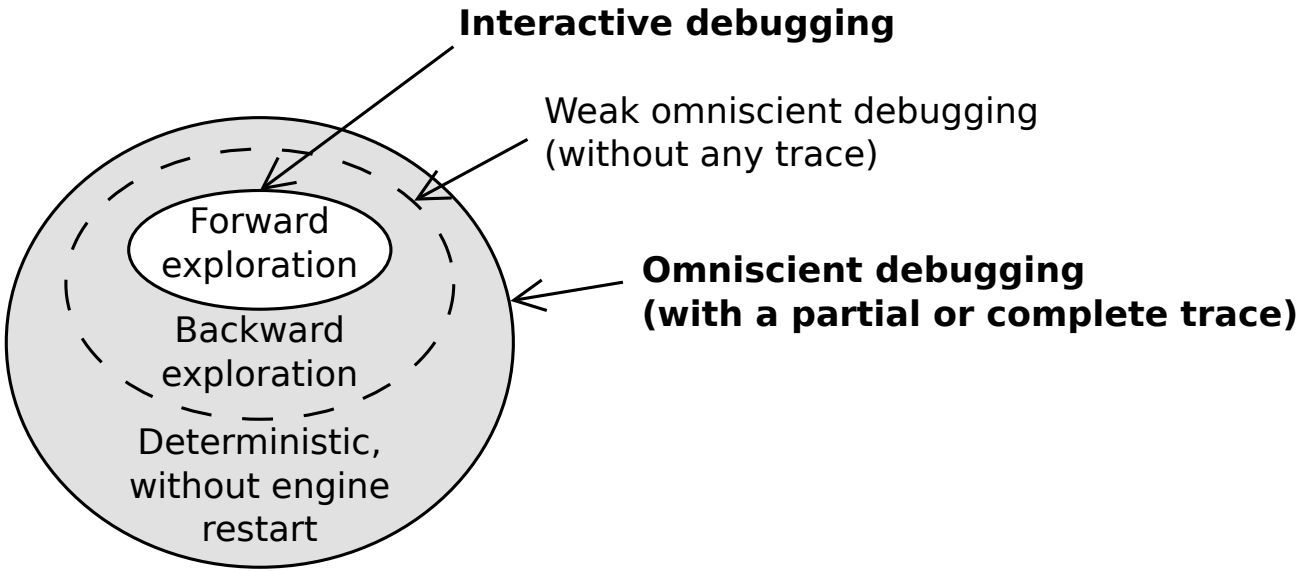



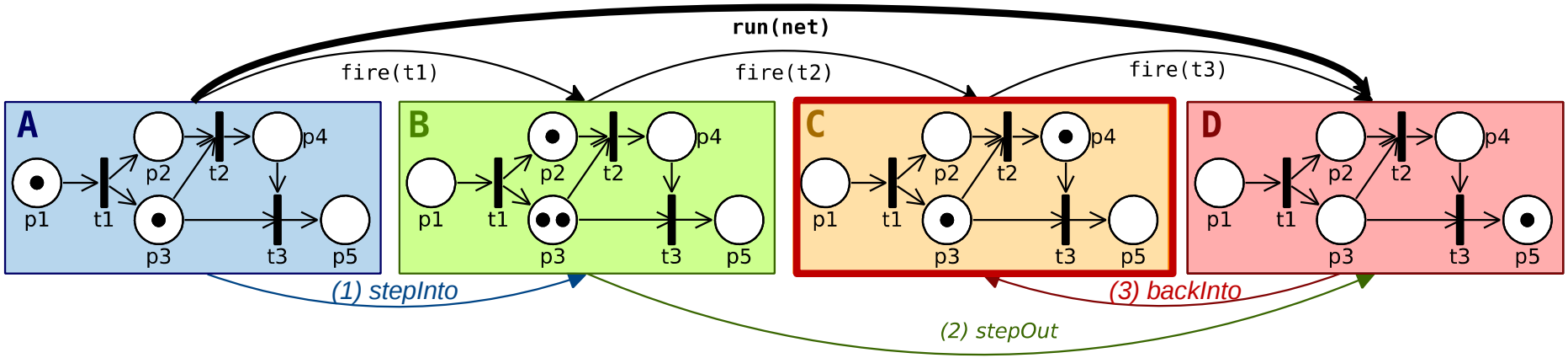
# Model omniscient debugging: terminology and scope



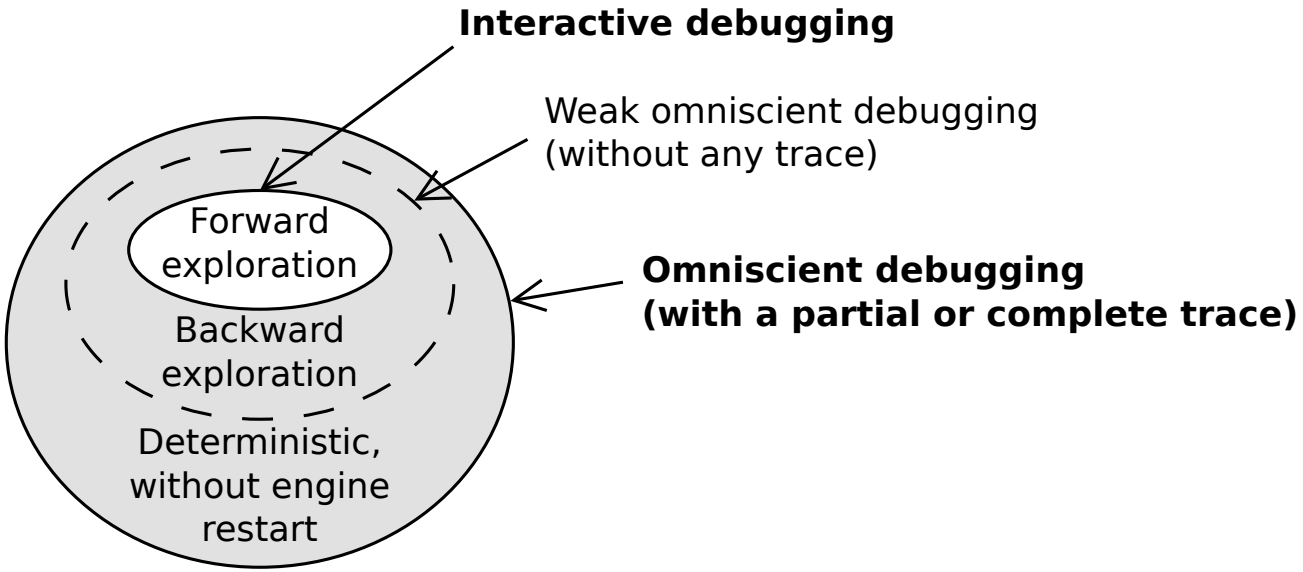



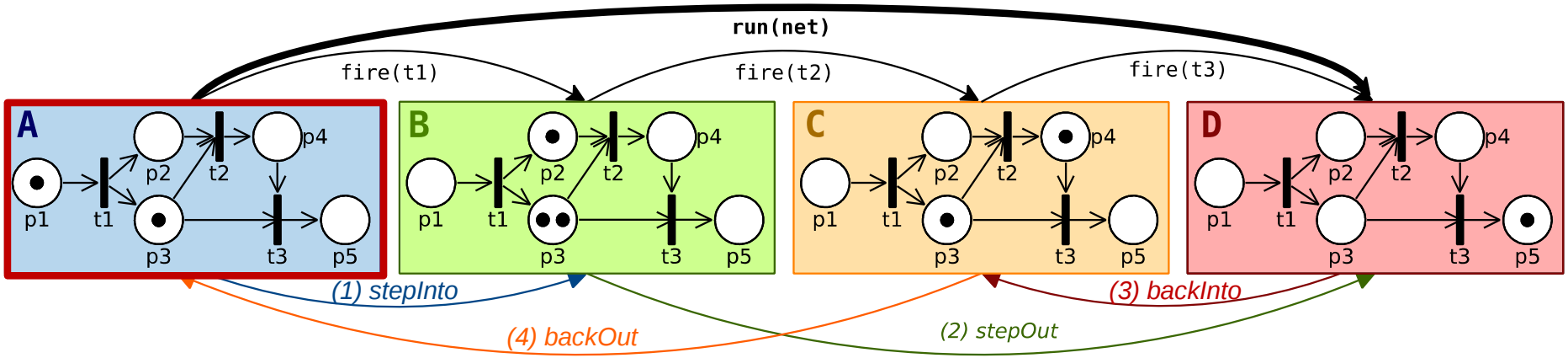
# Model omniscient debugging: terminology and scope



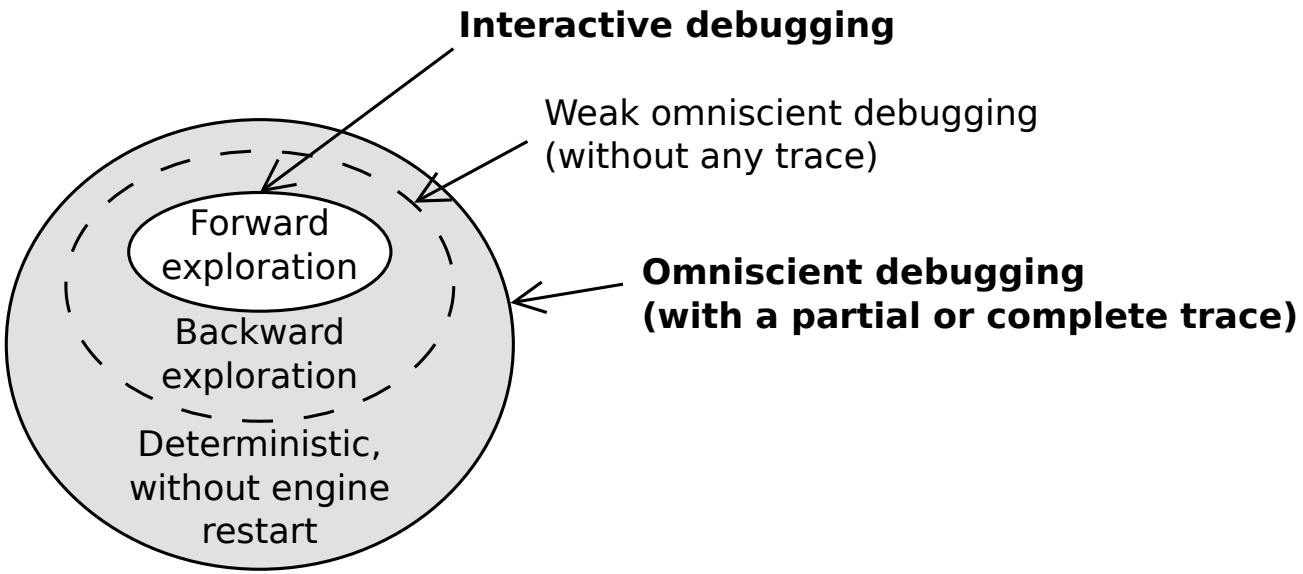



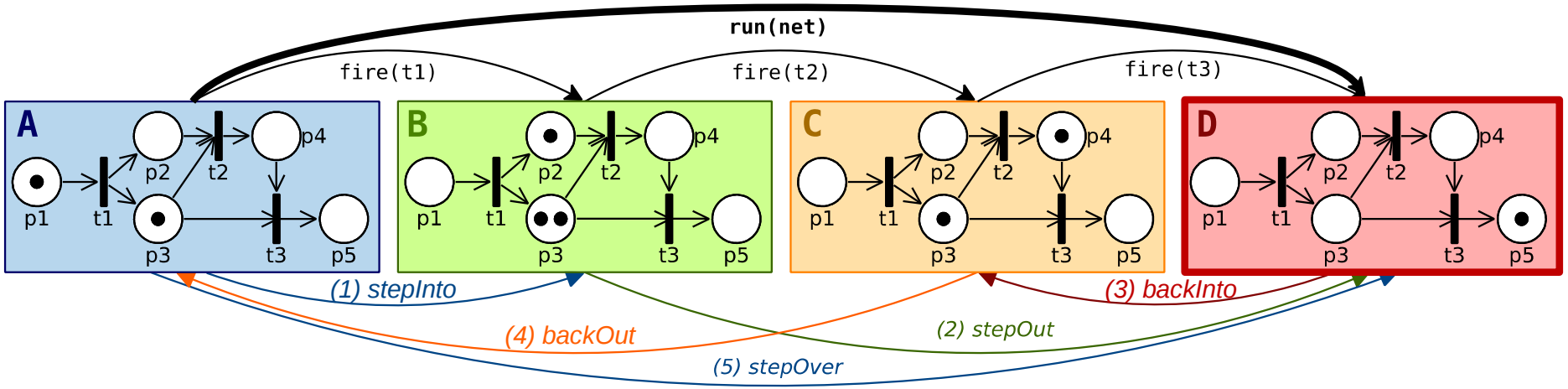
# Model omniscient debugging: terminology and scope



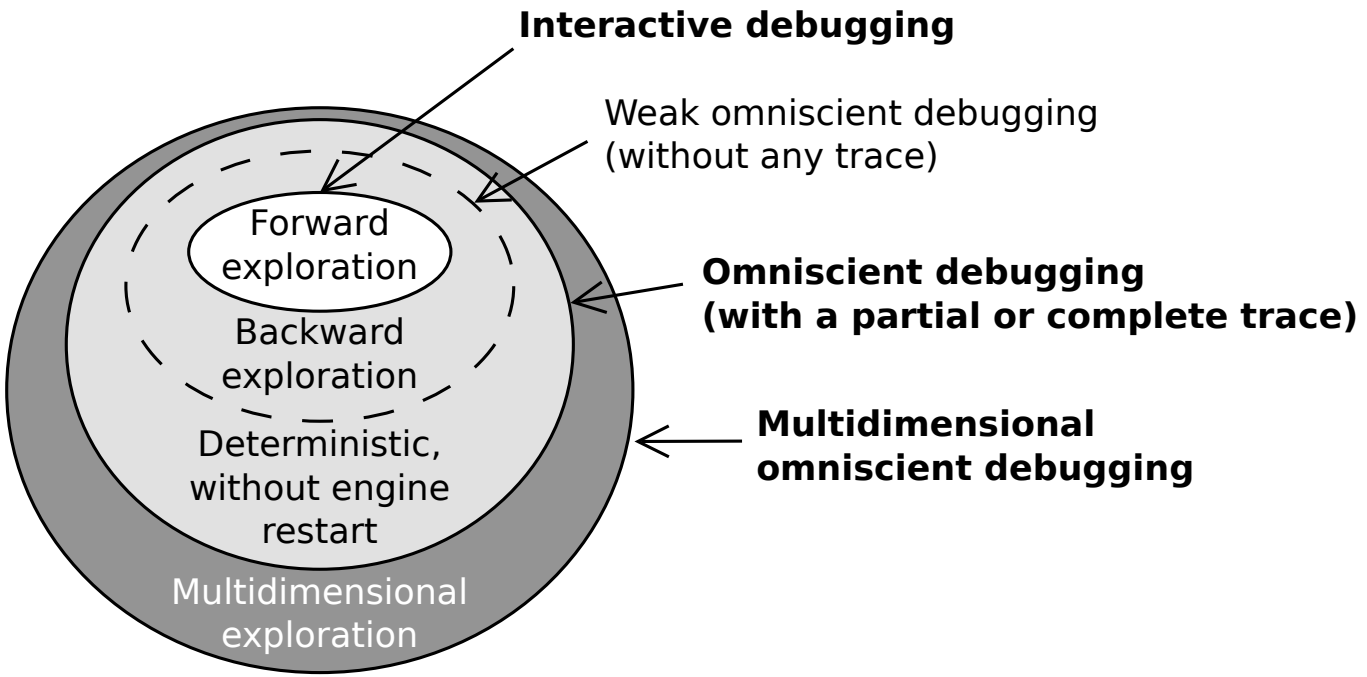



# Model omniscient debugging: terminology and scope



# Model omniscient debugging: terminology and scope



# Omniscient debugging for any xDSML?

- Generic omniscient debugging possible, based on steps
- Two key problems: **responsiveness** and **understandability** when debugging models from arbitrarily complex xDSML

## Approach

1. **Efficient generated domain-specific trace management facilities** for *responsiveness*
   - generated trace metamodel (using our contribution)
   - generated trace manipulations (e.g. "jump back")
   - *RQ2 and RQ3 can be evaluated*
2. **Advanced generic multidimensional omniscient debugging services** for *understandability*

# Omniscient debugging for any xDSML?

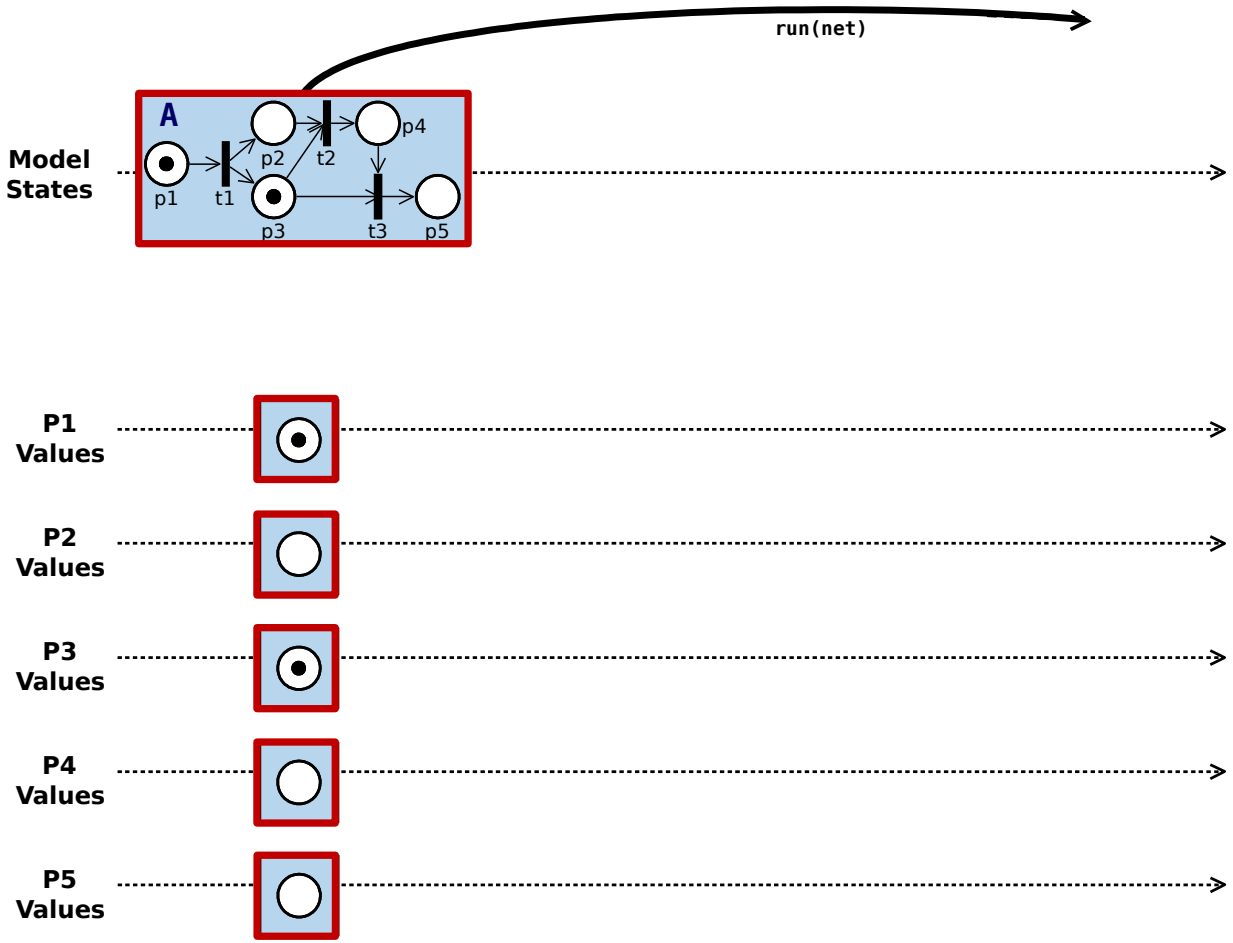
- Generic omniscient debugging possible, based on steps
- Two key problems: **responsiveness** and **understandability** when debugging models from arbitrarily complex xDSML

## Approach

1. **Efficient generated domain-specific trace management facilities** for *responsiveness*
   - generated trace metamodel (using our contribution)
   - generated trace manipulations (e.g. "jump back")
   - *RQ2 and RQ3 can be evaluated*
2. **Advanced generic multidimensional omniscient debugging services** for *understandability*

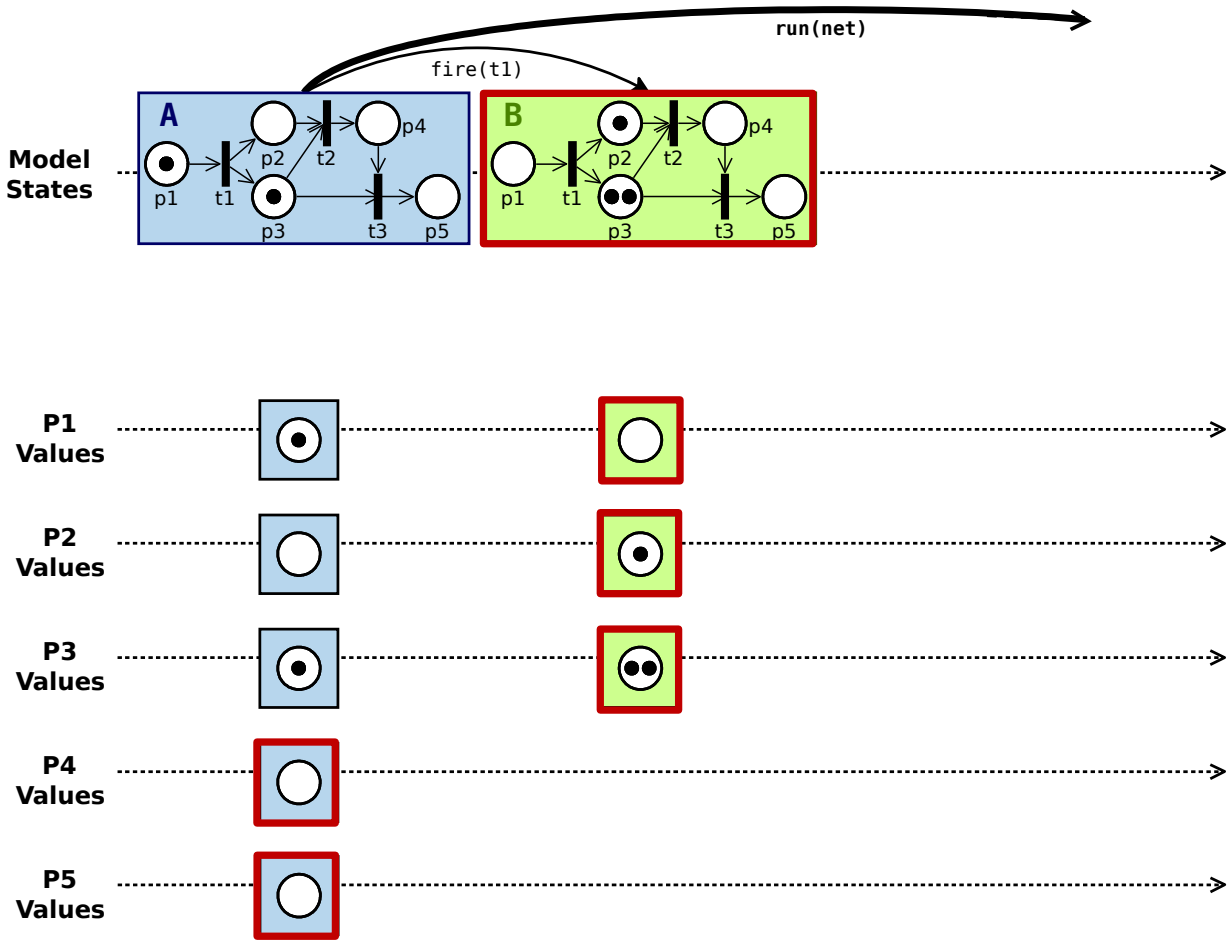
# Advanced omniscient debugging

- **New advanced debugging services** to navigate according to the different **dimensions** of the model



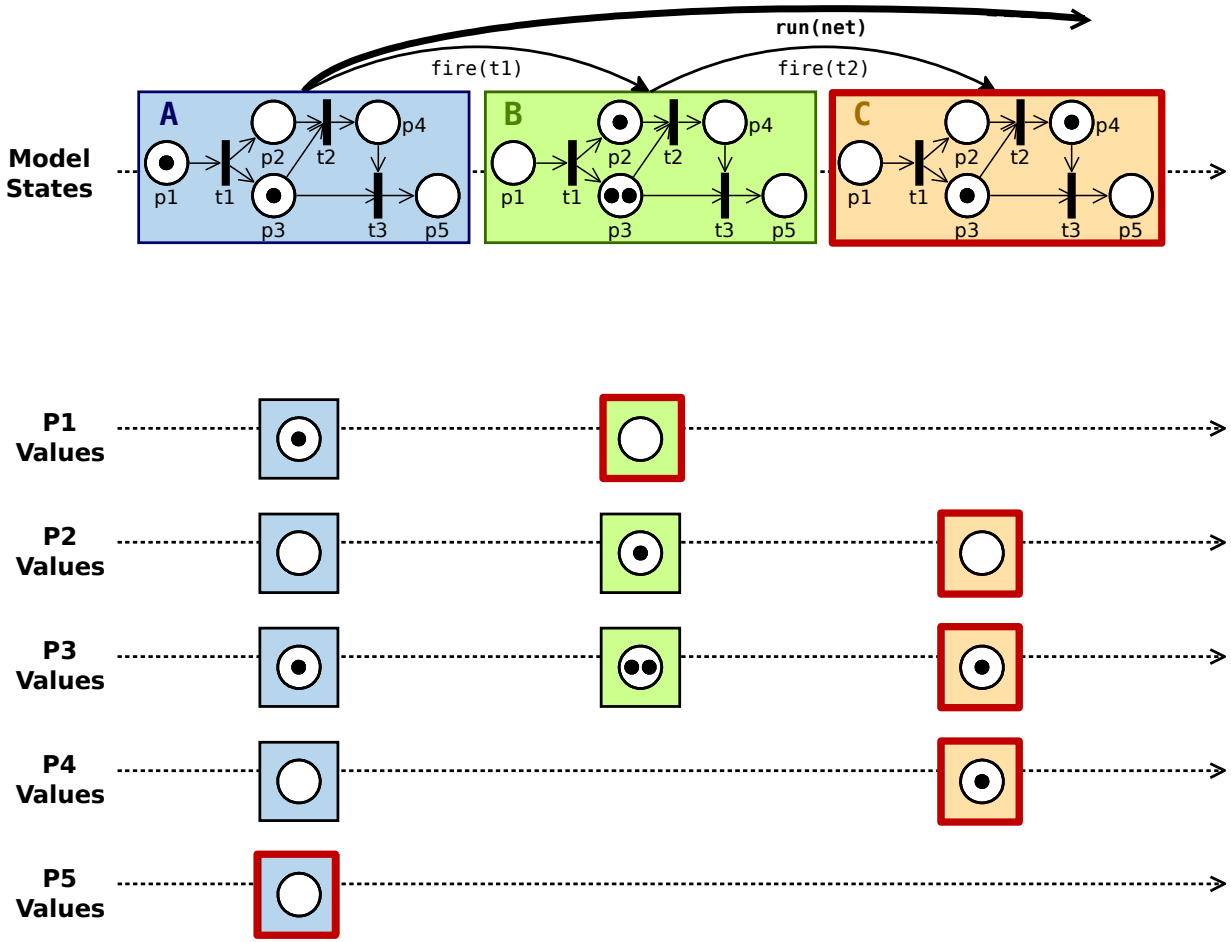
# Advanced omniscient debugging

- **New advanced debugging services** to navigate according to the different **dimensions** of the model



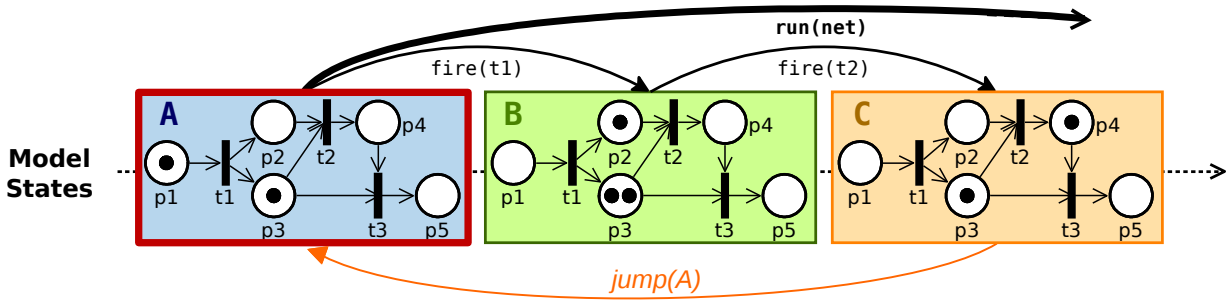
# Advanced omniscient debugging

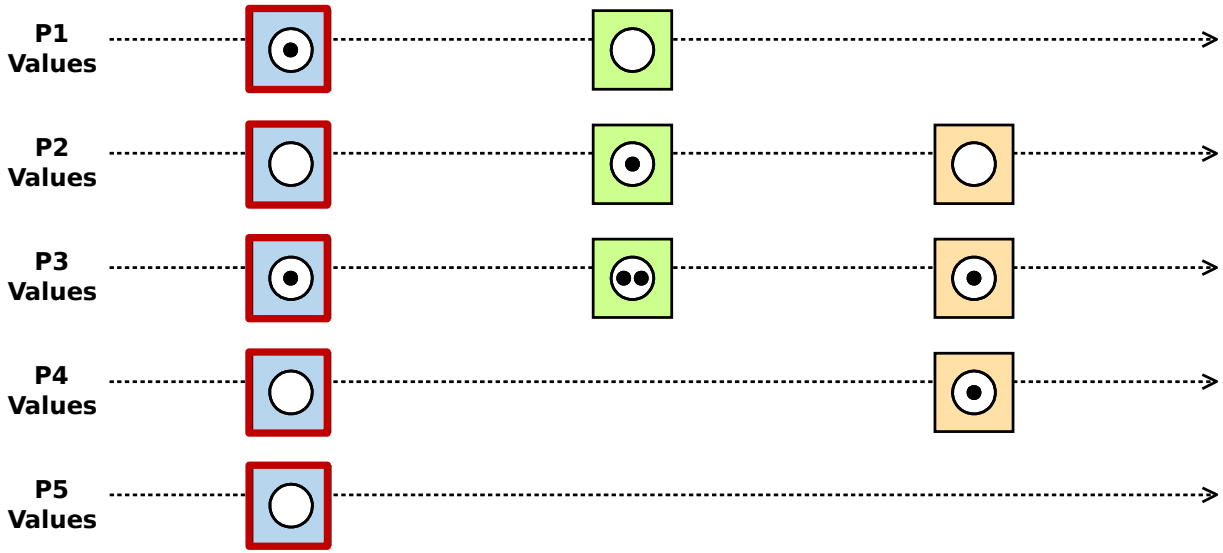
- **New advanced debugging services** to navigate according to the different **dimensions** of the model



# Advanced omniscient debugging

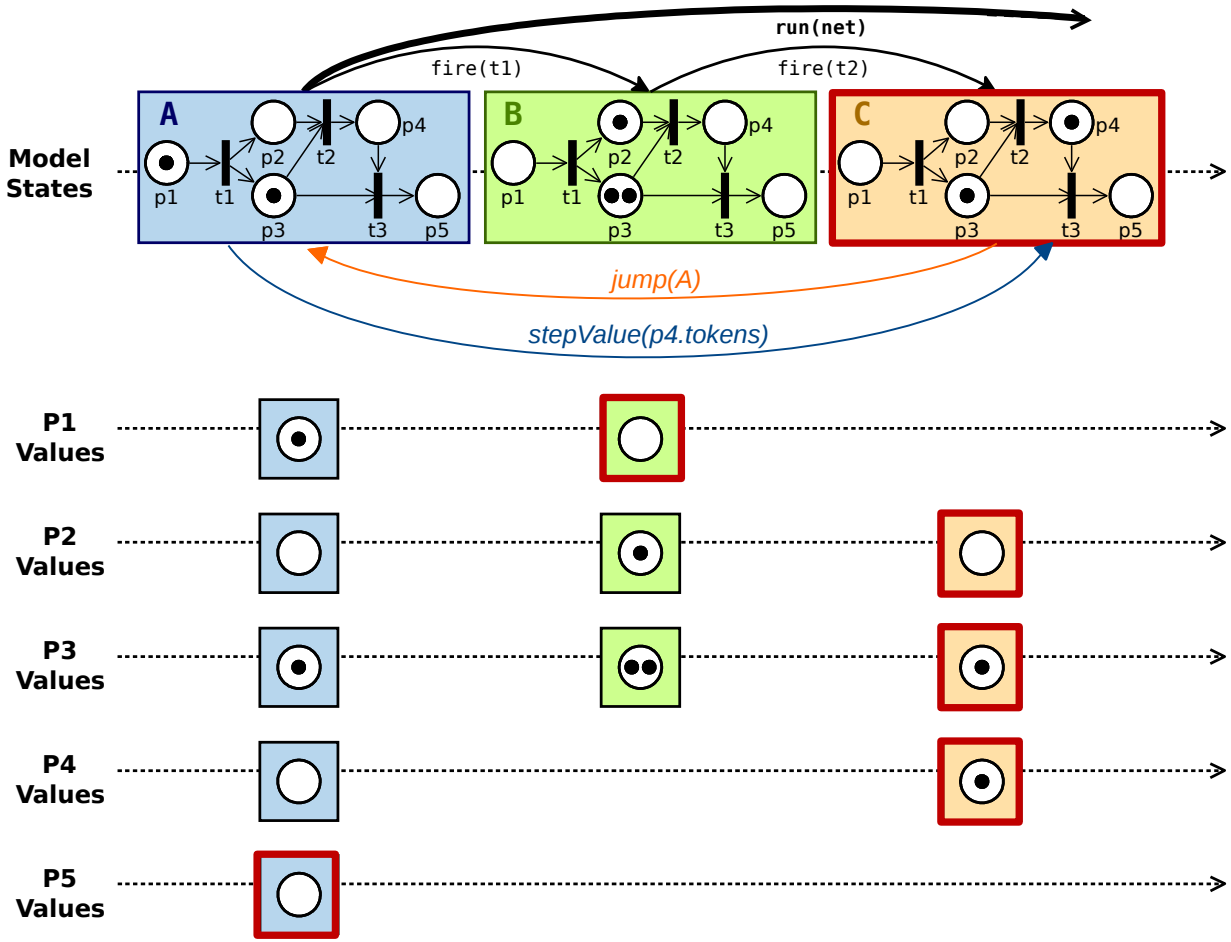
- **New advanced debugging services** to navigate according to the different **dimensions** of the model



# Advanced omniscient debugging

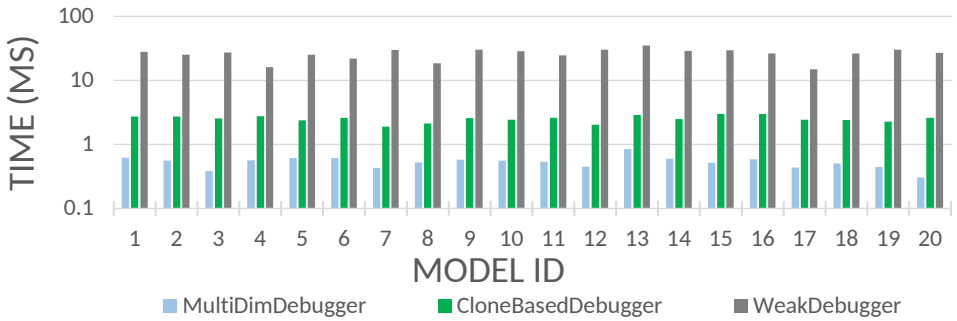
- **New advanced debugging services** to navigate according to the different **dimensions** of the model



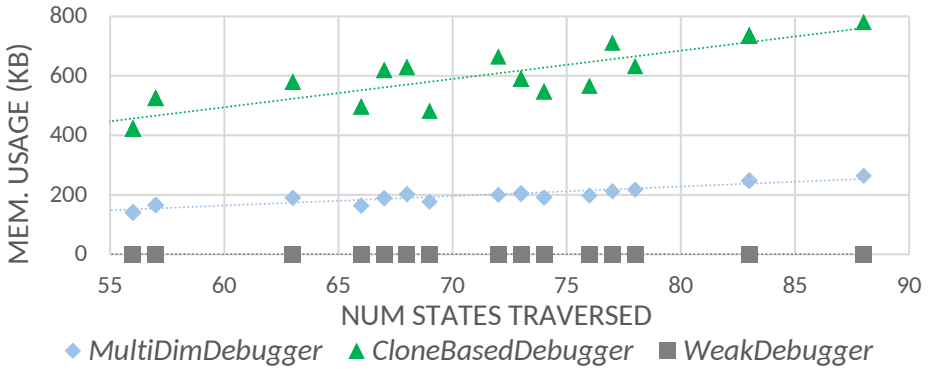
# Evaluation of responsiveness

- **Same case study:** fUML models from [Maoz et al.'11]
- Comparison of our advanced debugger with two others: weak (no trace) and clone-based (generic trace)

*"Jump back" execution time (RQ2)*




*Memory consumption (RQ3)*



# Note regarding evaluation of time

RQ2 (time) was evaluated in two different ways:

- *Semantic differencing*: the **whole trace** is read and compared
- *Omniscient debugging*: a **single state** is read to jump back

Results are improved in both cases:

- Thanks to the **dimensions** when reading the whole trace
- Thanks to the **precise capture of the execution concepts**, when reading a single state

# Outline

# Conclusion

*How to provide efficient execution trace management facilities for any xDSML?*

- Contribution: **new generative approach** to manage execution traces of any xDSML
- Validation in a **realistic context** with applications to two existing V&V techniques using a real-world xDSML
- Results show that:
  - RQ1: **Reduces complexity** of domain-specific manipulations
  - RQ2: **Reduces processing time** by 75%
  - RQ3: **Reduces memory consumption** by 75%

# Short-term perspectives

- **Customization of generated trace metamodels** to better fit expected trace manipulations, e.g. ignore unused parts of the metamodel
- **Represent execution branches** sharing a common prefix, within a single execution trace
- **Domain-specific property language** to define temporal properties over traces (cf. [Maoz et al.'14])
- **Domain-specific debugging services** using the underlying domain-specific trace metamodel (cf. [Chiş et al.'15])

# Long-term perspectives

## Execution traces as core modeling artifacts

- **"Live modeling"**: instant visualization of the impact of a change in a model regarding its behavior
- Versionning of the behavior of models: "trace of traces"

## Generating *everything* for an xDSML

- Generating domain-specific mutation operators
- Generating a DSML to define test suites for an xDSML
- ...

# Publications

International conferences:

- **Model omniscient debugging**:
  E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry. "Supporting Efficient and Advanced Omniscient Debugging for xDSMLs". SLE 2015.
- **Trace metamodel generation**:
  E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. "A Generative Approach to Define Rich Domain-Specific Trace Metamodels". ECMFA 2015.
- **Model cloning**:
  E. Bousse, B. Combemale, and B. Baudry. "Scalable Armies of Model Clones through Data Sharing". MODELS 2014.

+ 4 international workshops papers

# Appendix

# Outline

# Types of semantics (I)

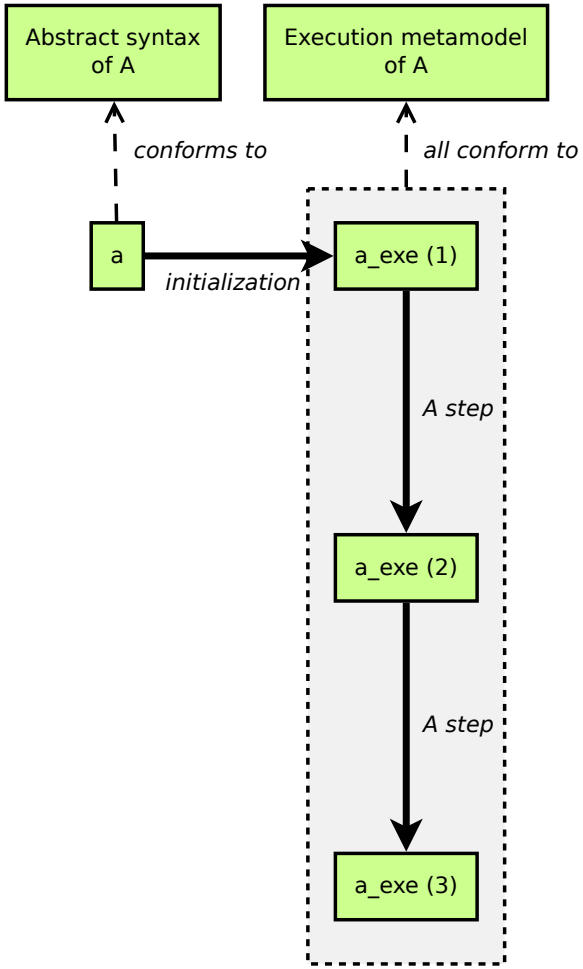



Figure: Operational semantics

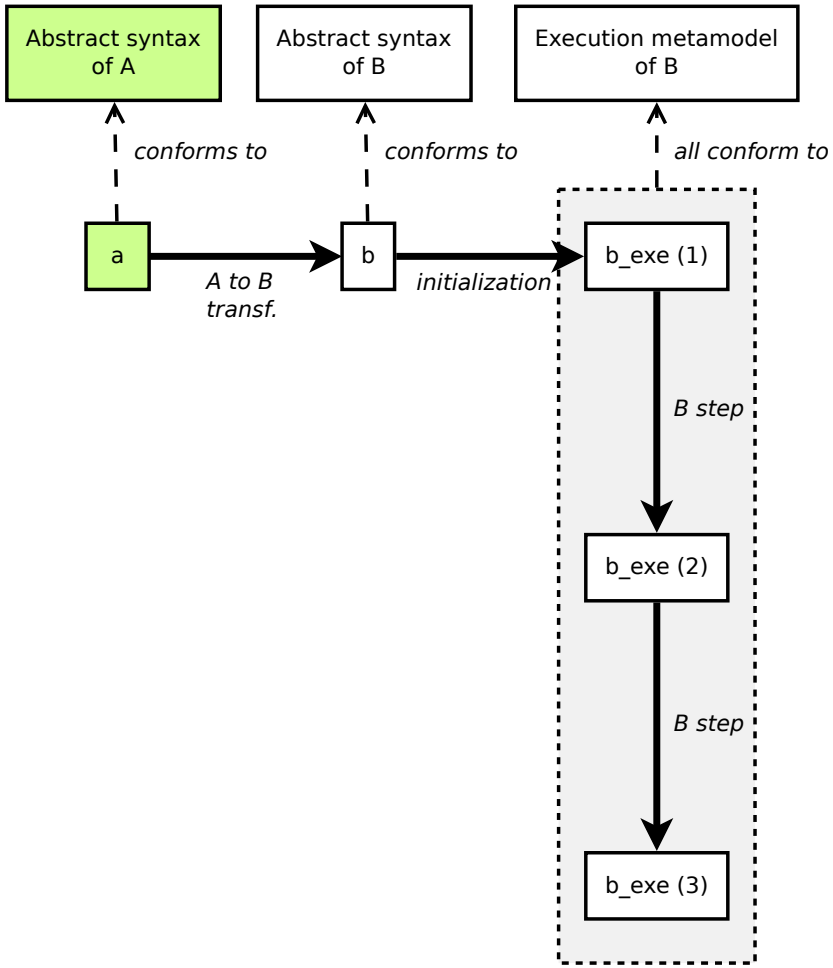



Figure: Translational semantics

# Types of semantics (II)

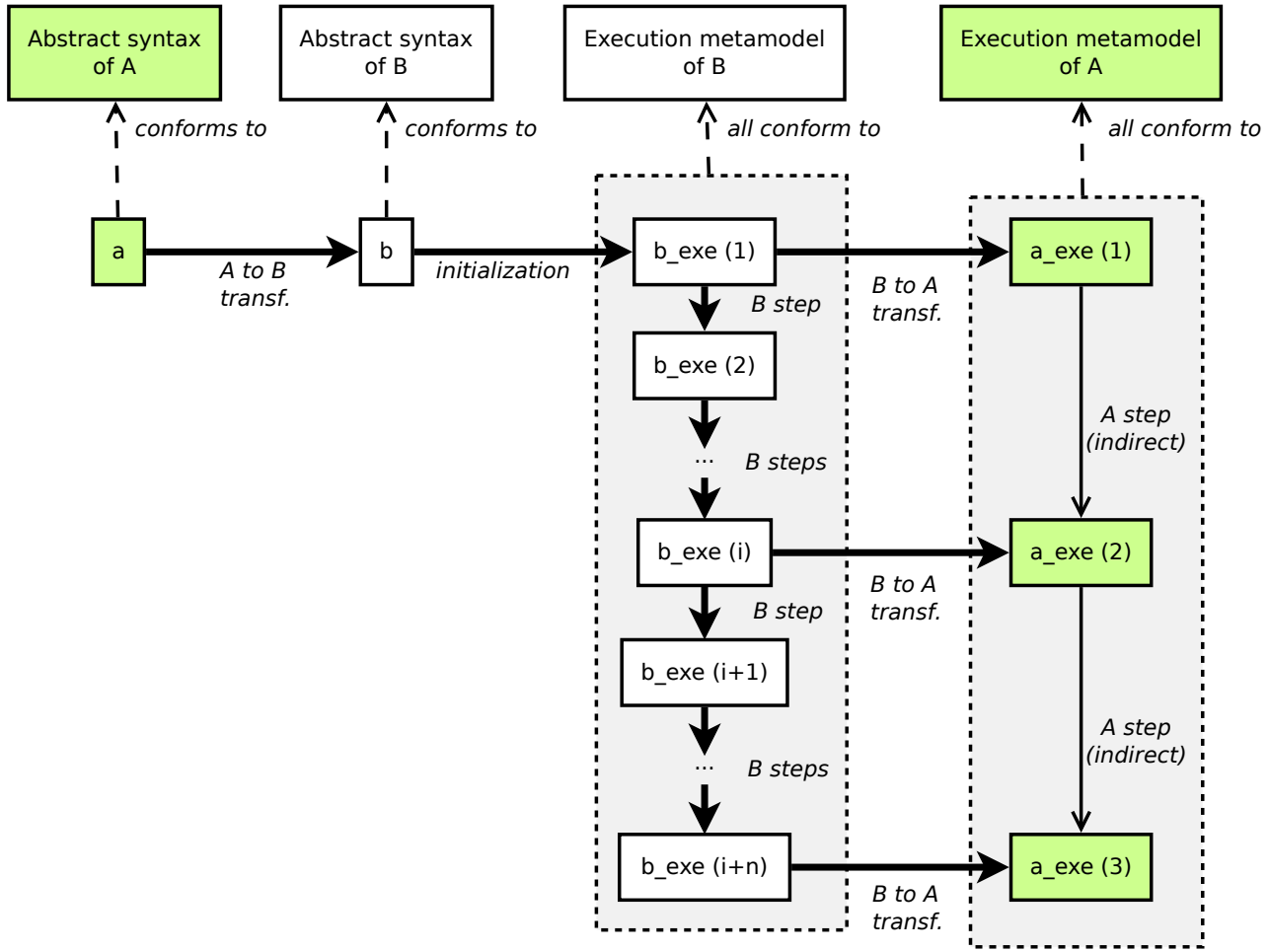



Figure: Translational semantics with back-annotation

# Implicit steps

```
1  def void run() {
2      ... // code with model change (1)
3      someTransition.fire()
4      ... // code with model change (2)
5  }
```

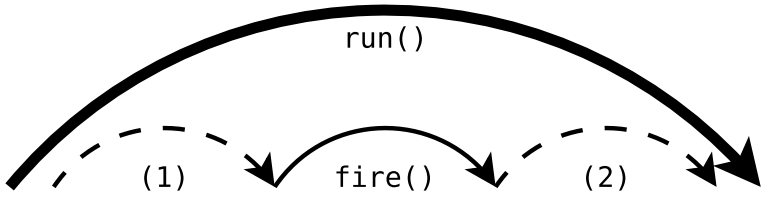



Figure: Implicit step illustration

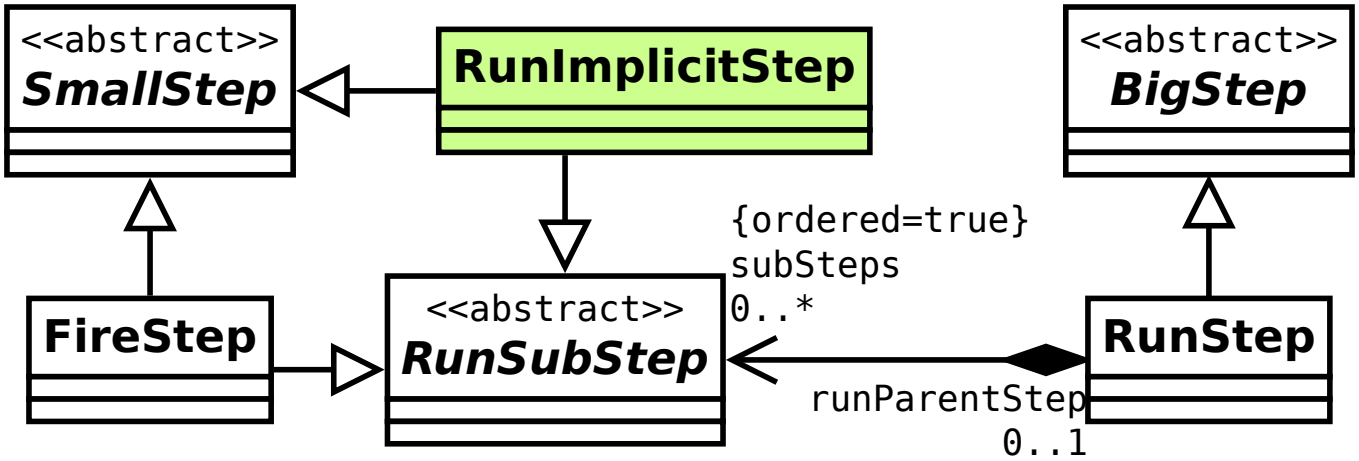
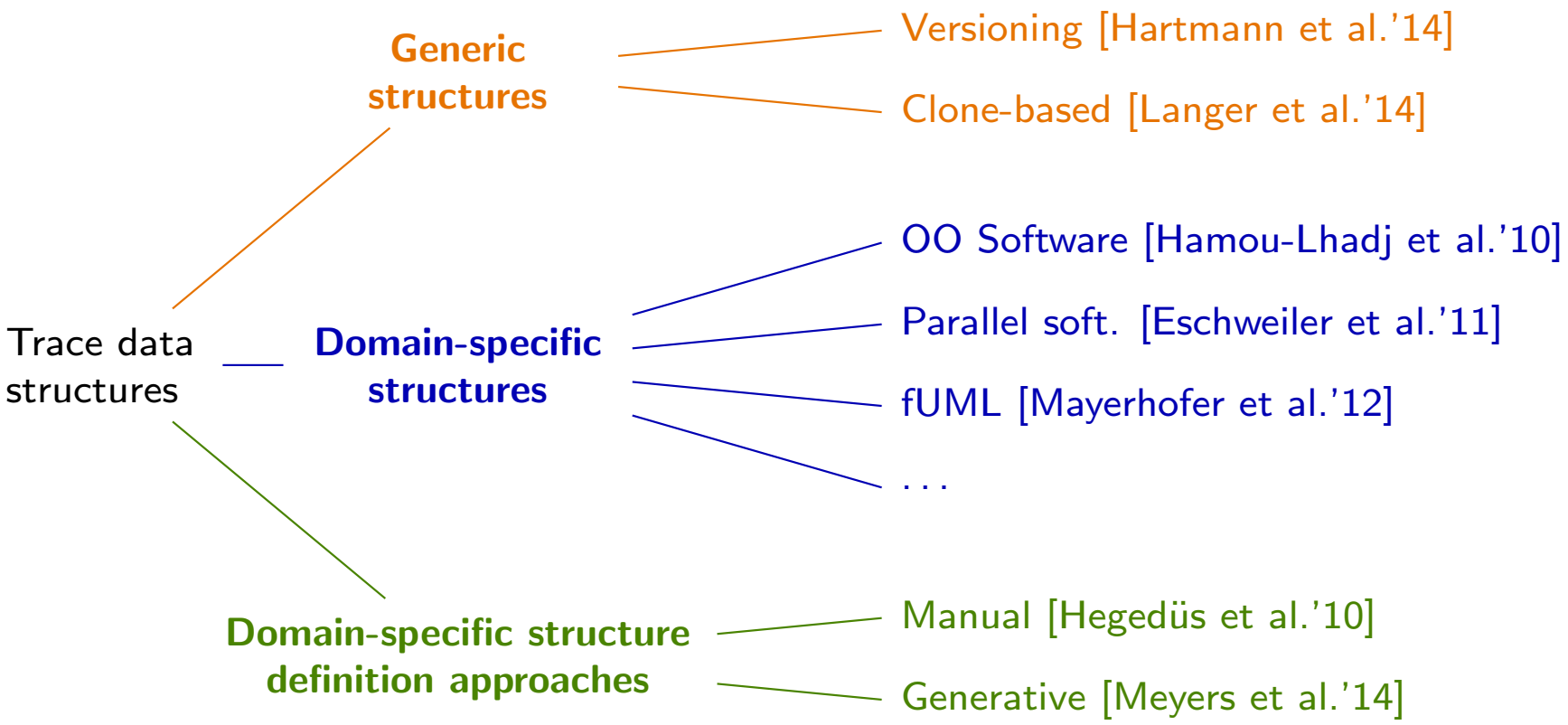



Figure: Petri net implicit step class

# State of the art (summary)

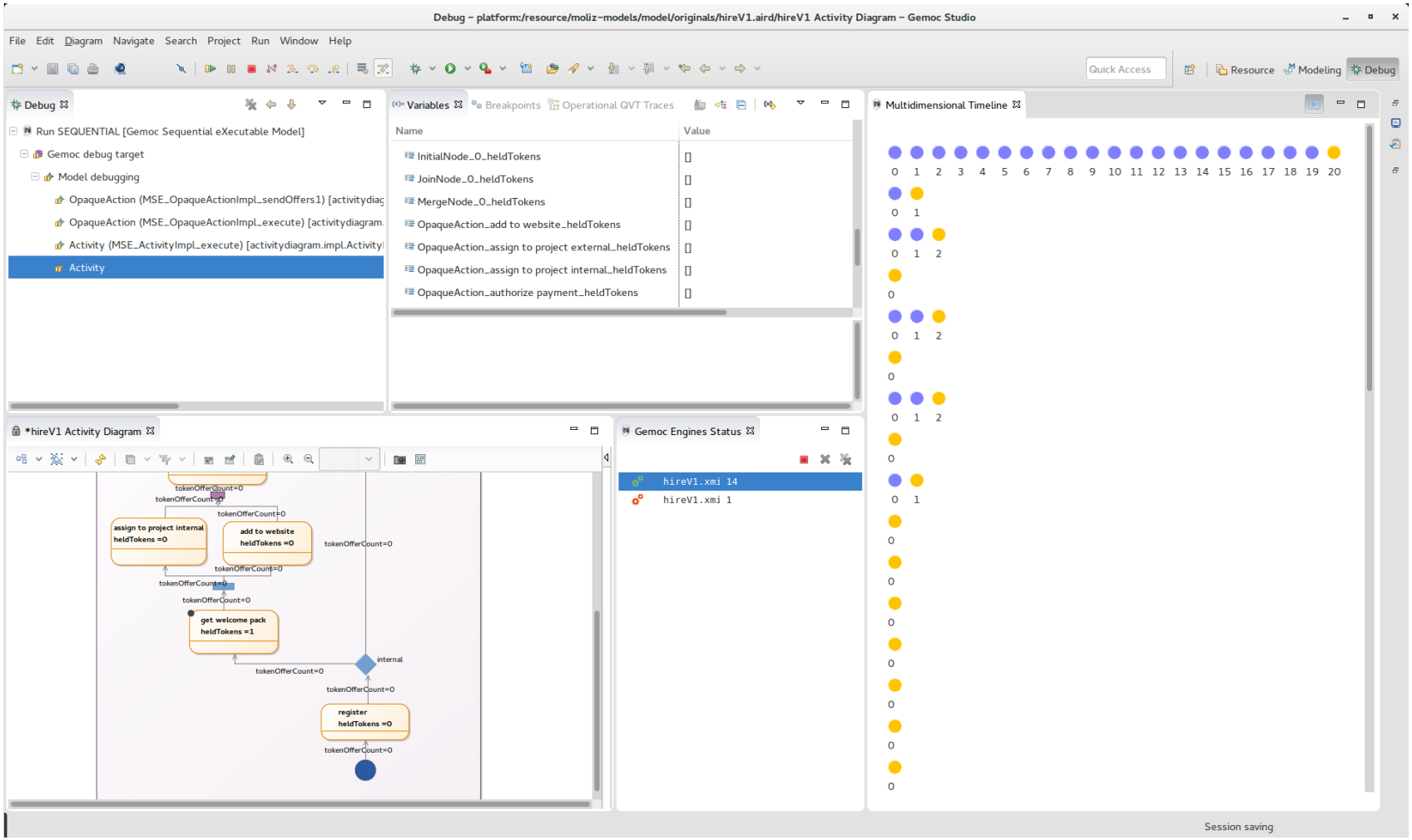


# State of the art (table)

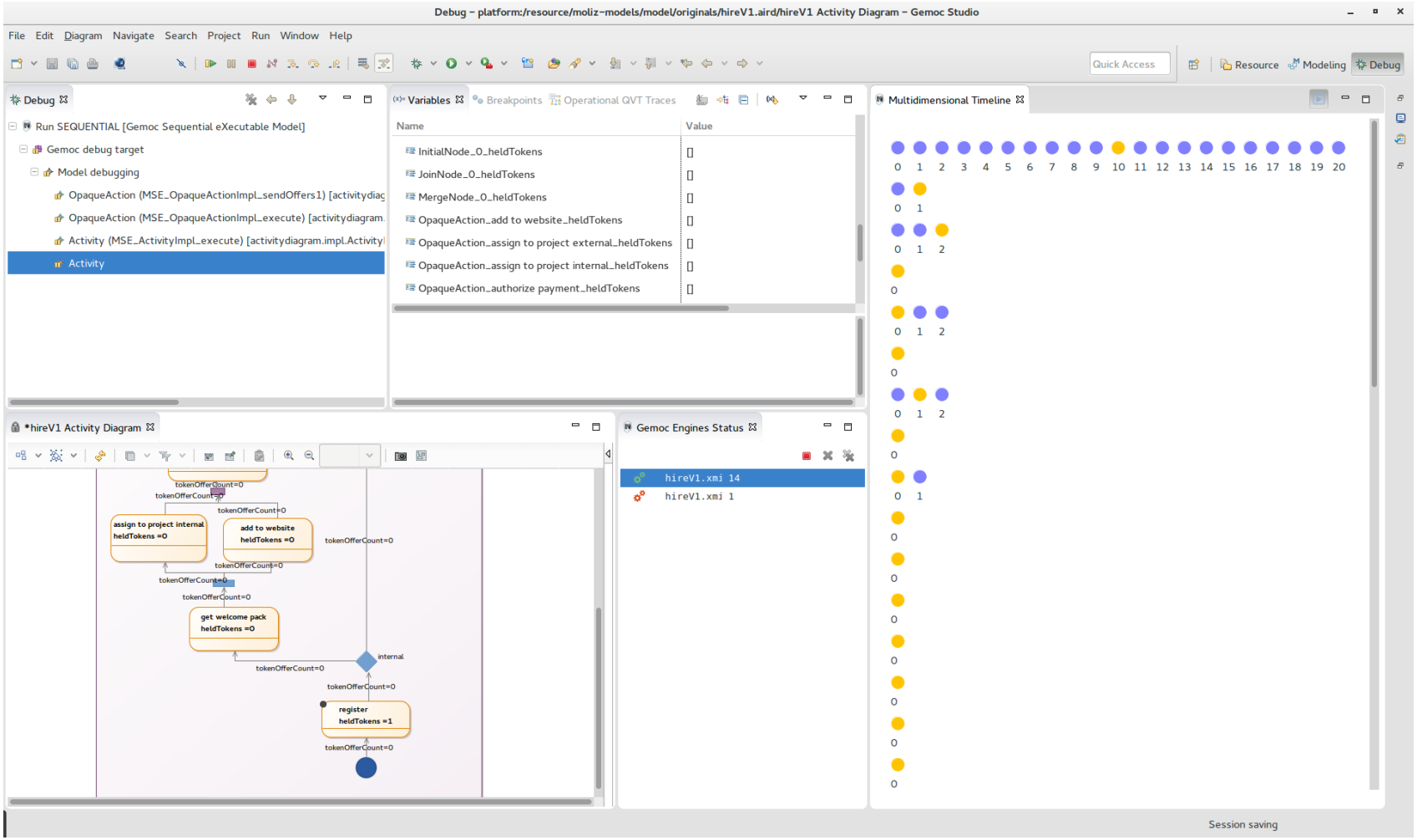
| Name | Type | Ev./St. | Concerns |
|---|---|---|---|
| Open Trace Format 2 [56] | ASCII format | Both[a] | Parallel software |
| MPI Trace Format [3] | Metamodel | Events | HPC [b] |
| Compact Trace Format [77] | Metamodel | Events | Software |
| KPTrace [146] | ASCII format | Events | Operating systems |
| CUBE4 [67] | Binary format | Both[c] | Distributed software |
| UML Testing Profile [75] | Metamodel | Events | Software (UML) |
| fUML [113] | Metamodel | Events | fUML |
| Scenario-Based Traces [111] | ASCII format | Events | Sequence charts |
| Timesquare [46] | Metamodel | Events | Time, Timesquare |
| Gen. Sem. Diff. [99] | Metamodel | Both | Generic |
| KMF Versioning [81] | Other | States | Generic |
| Pablo SDDF [4] | ASCII/Binary | Both | Self-defining |
| Pajé [139] | ASCII format | Both | Self-defining |
| SOC-Trace project [124] | Metamodel | Events | Self-defining |
| Common Trace Format [47] | ASCII/Binary | Events | Self-defining |
| TopCased [32, 41] | Approach | Events | Domain-specific |
| Hegedüs et al. [83] | Approach | Both | Domain-specific |
| Promobox [116] | Generative App. | Both | Domain-specific |
| Filmstrip models [69, 85] | Generative App. | Both | Domain-specific |

Figure: Execution trace data structures

# Screenshot GEMOC debugger (during execution)



# Screenshot GEMOC debugger (after jump back)



# fUML case study

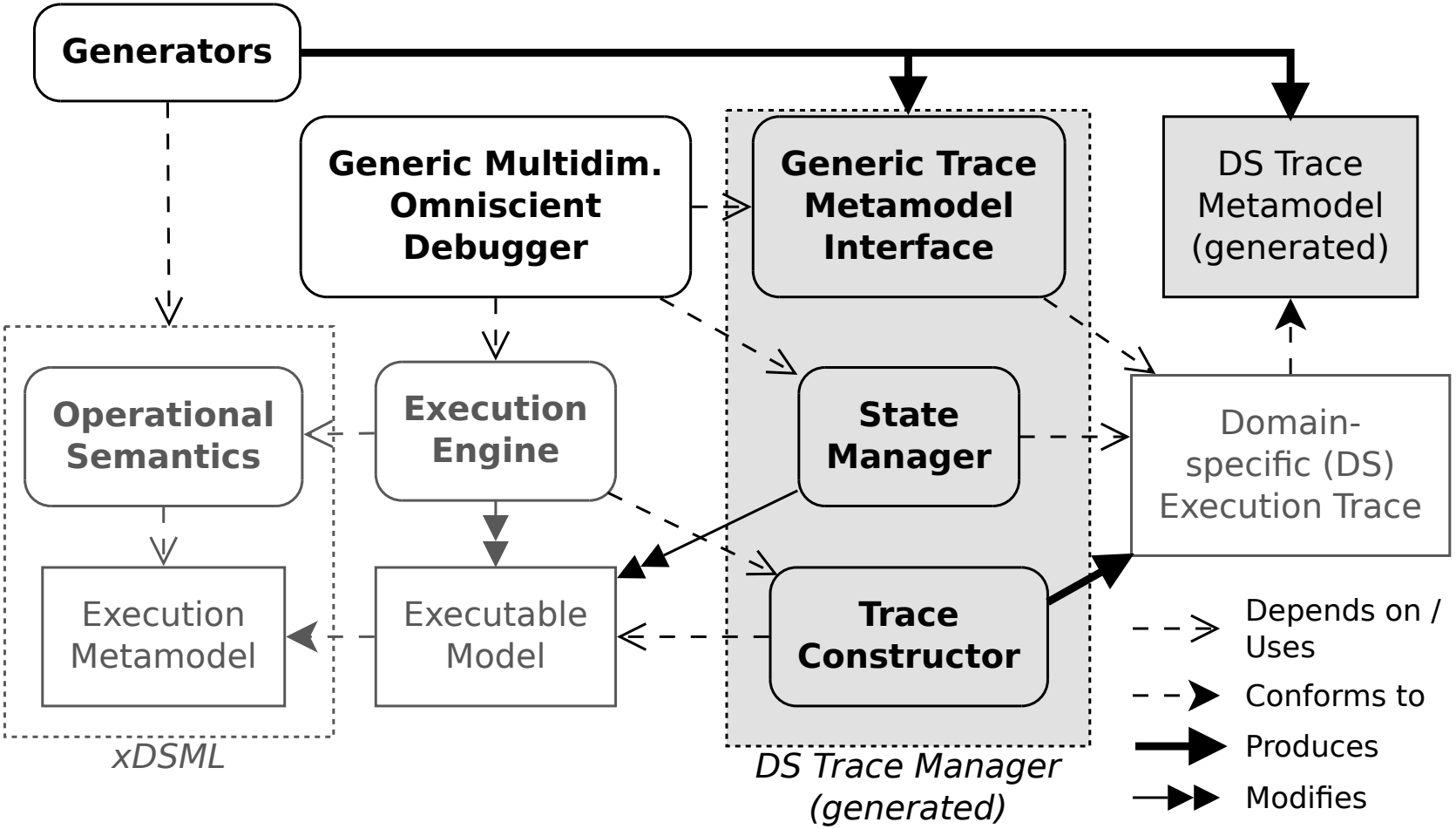
## fUML xDSML

- subset of UML
- 57 classes

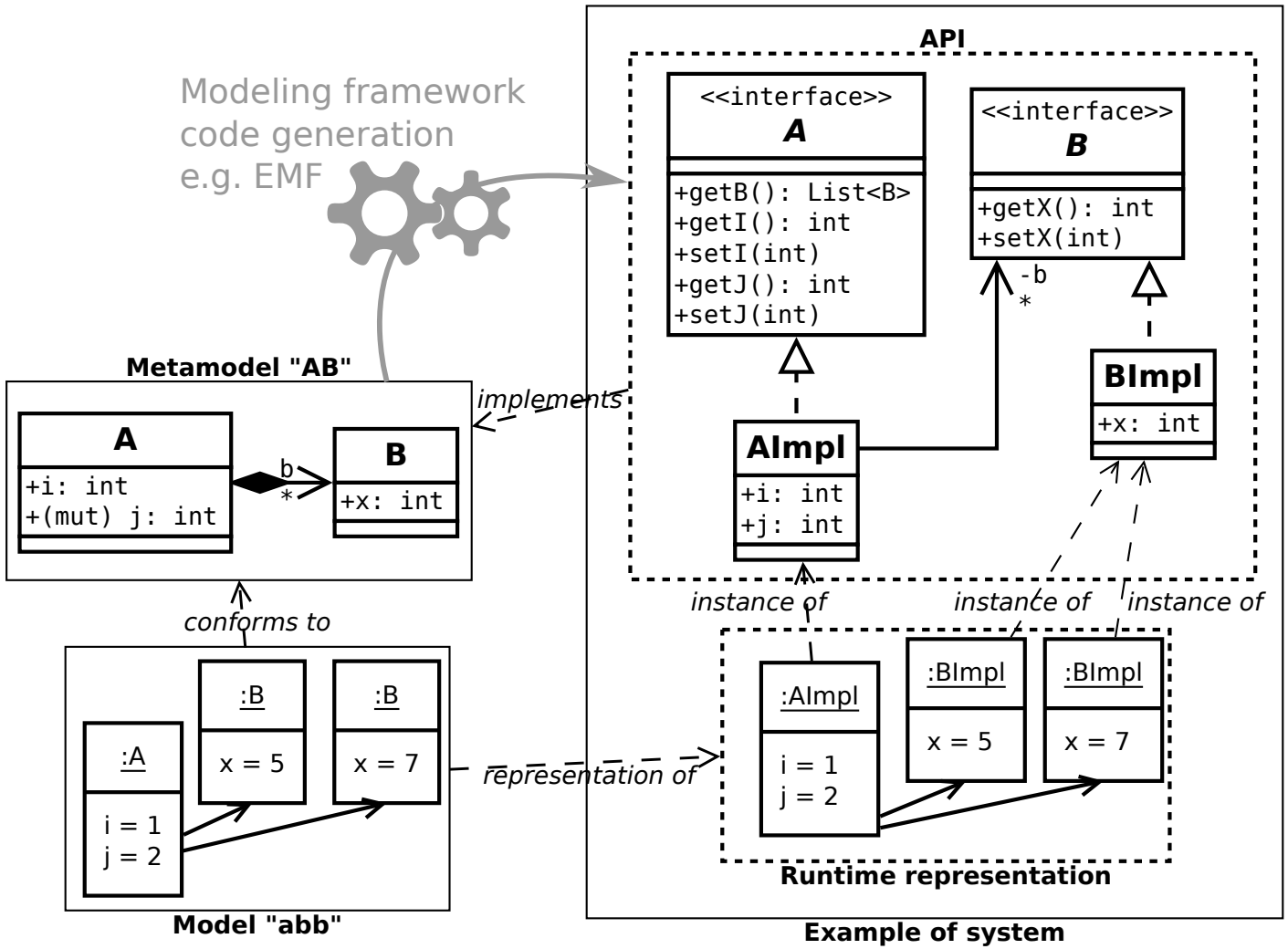
## fUML Models

- Case study of Maoz et al. to evaluate a semantic differencing operator
- Models drawn from industrial sources
- Multiple sequences of models, each sequence with a change
- 40 models considered, sizes range from 36 to 51 objects
- Source: `http://www.se-rwth.de/materials/semdiff/`

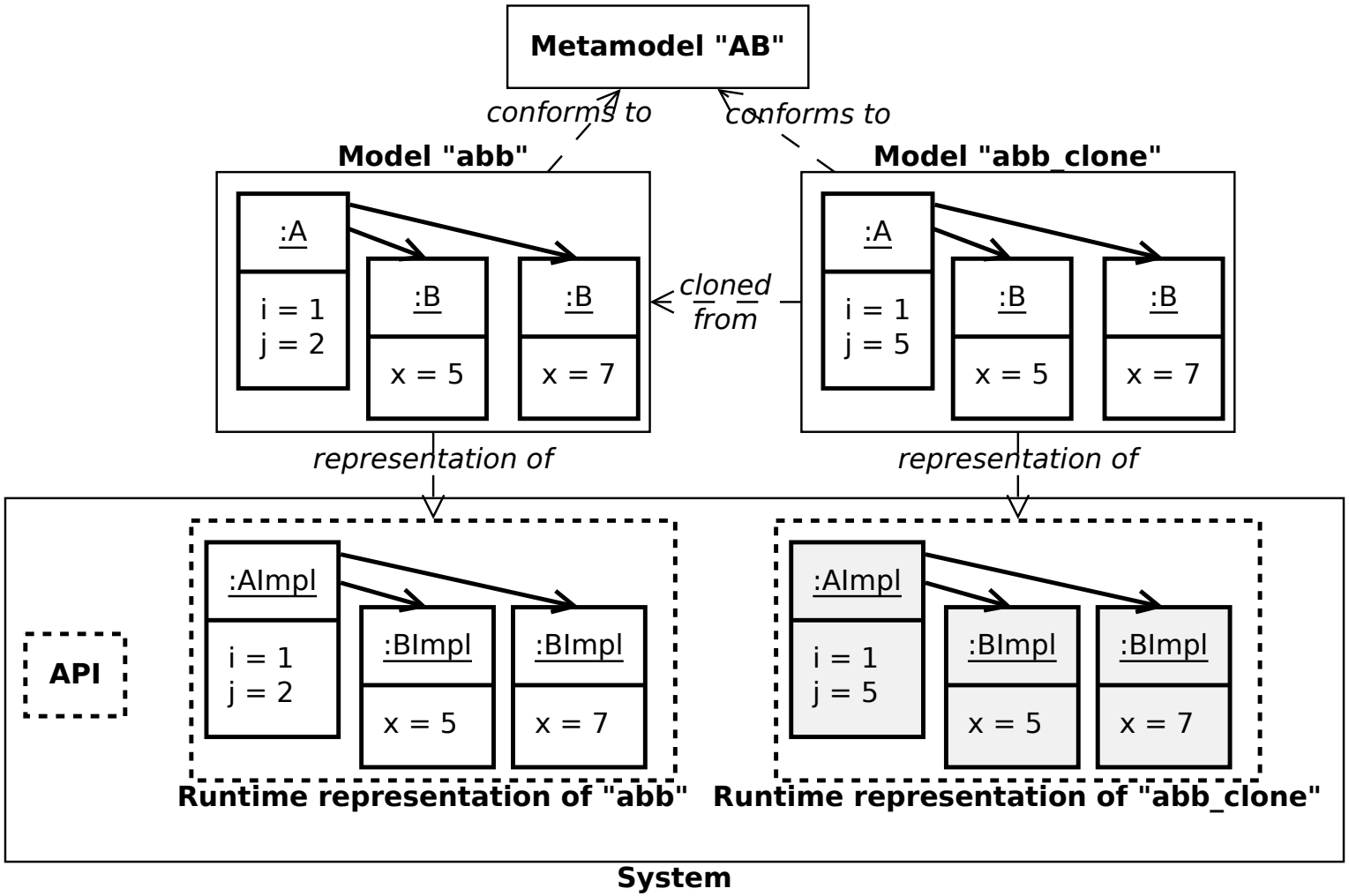
# Omniscient debugger architecture



# Example: a metamodel, a model, and runtime counterparts



# Example: a model clone obtained using *deep cloning*



# Data sharing: existing approaches

Idea for Req #1: considering that only a subset of a model changes during its lifecycle, **avoid data redundancy among clones**

## Dynamically: copy-on-write (aka lazy copy)

*Create virtual copies, and create real copies on write accesses.*

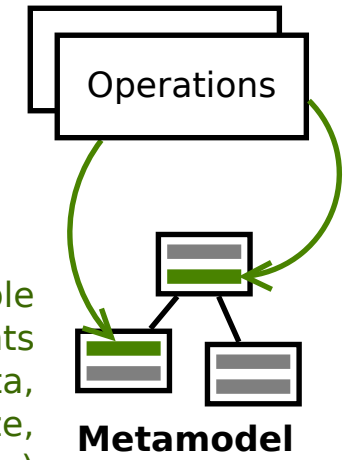
- either using a specific API/entry-point, **breaks Req#3**
- or in a transparent way, but managing consistency depends on the implementation language (e.g. Java is pass-by-value)
- Copies are done during manipulations, **may break Req#2**

## Statically: flyweight design pattern

*Objects are designed to be used in multiple contexts.*

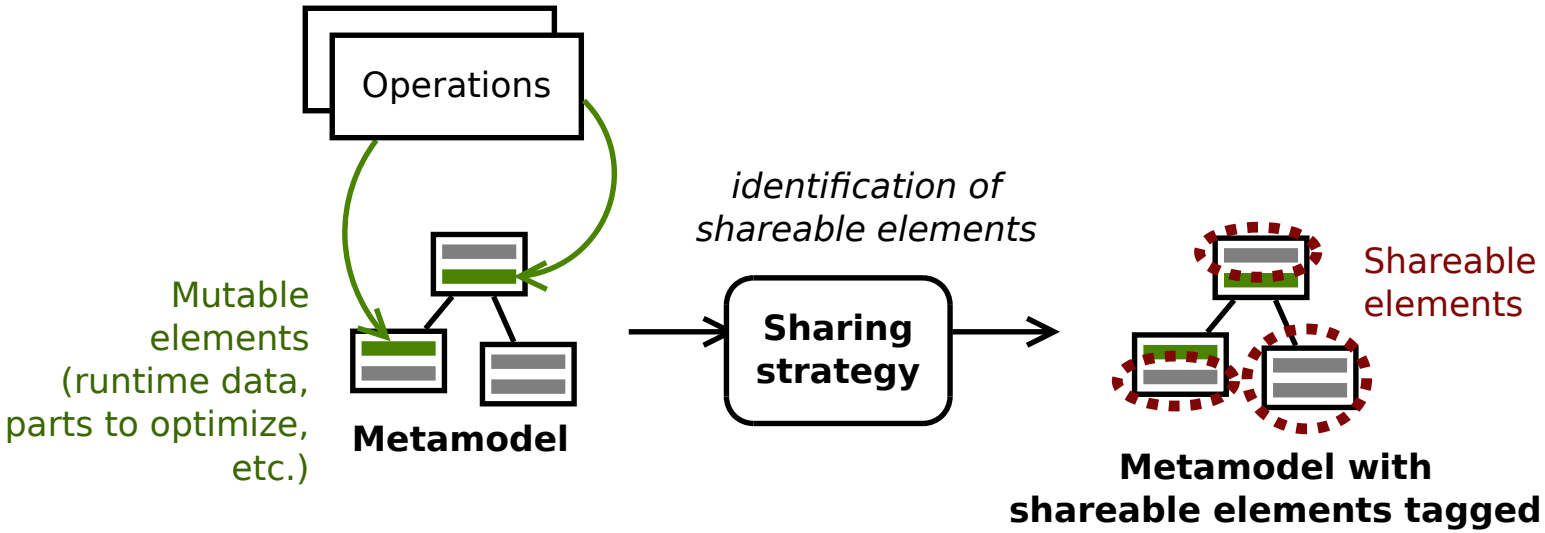
- Requires the passing the extrinsic state (ie the mutable part) of the object as a parameter, for all its operations, **breaks Req #3**

# Approach: *static* identification of safely shareable parts

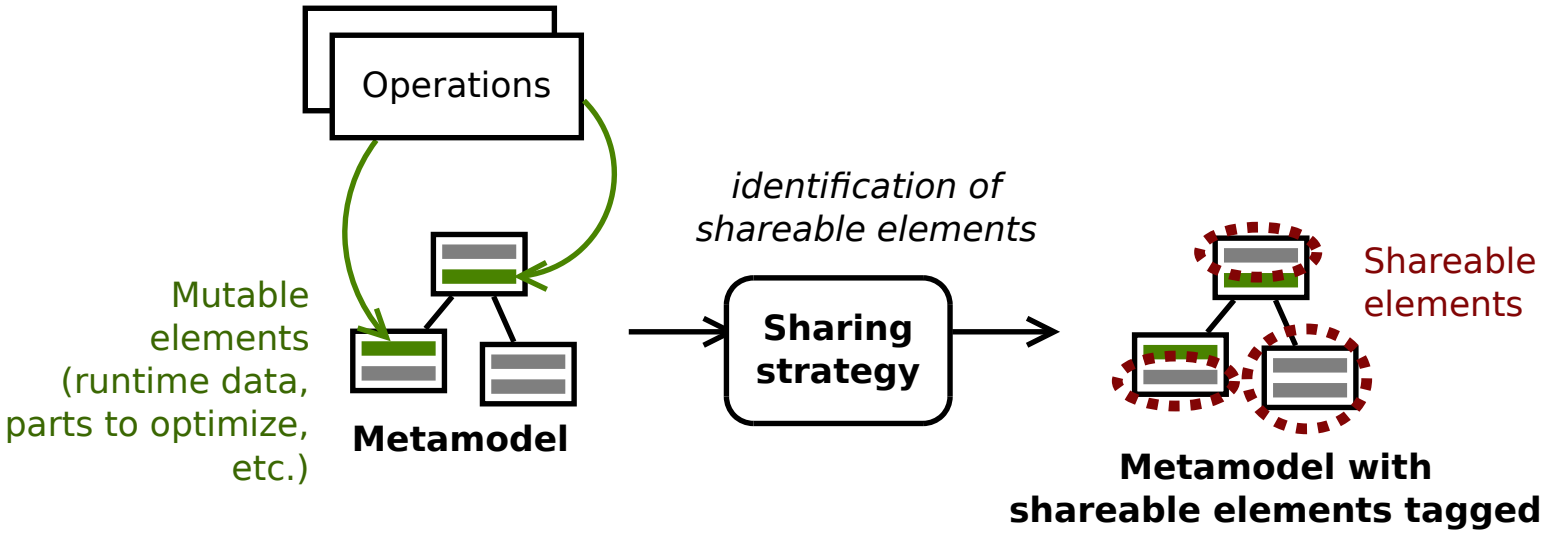


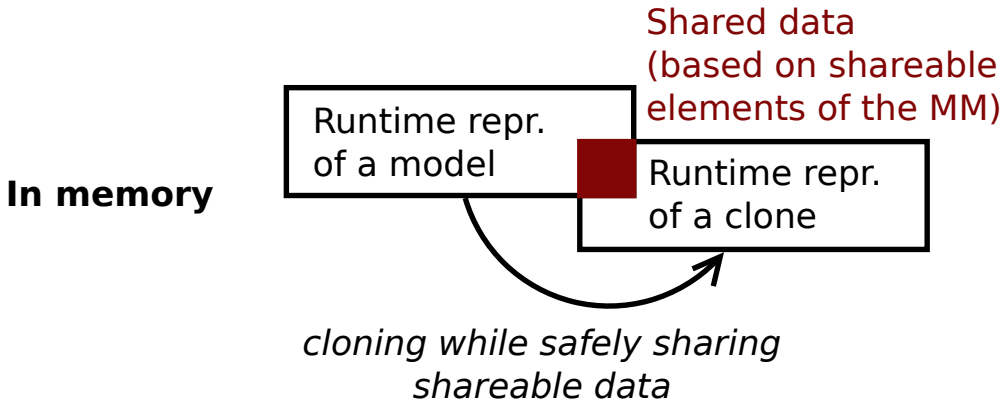

Mutable elements (runtime data, parts to optimize, etc.)

# Approach: *static* identification of safely shareable parts

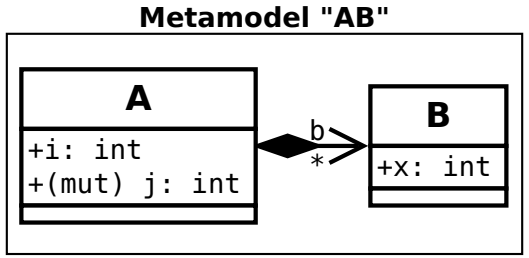


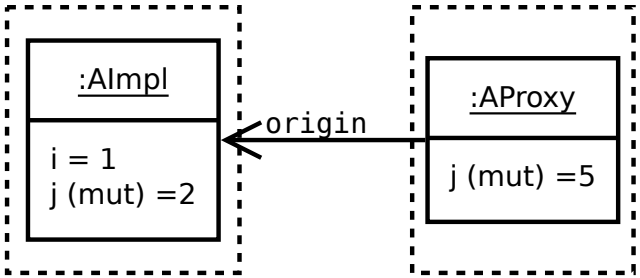
# Approach: *static* identification of safely shareable parts



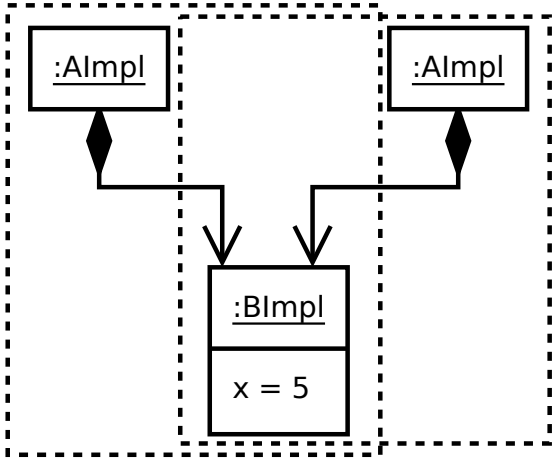



# Shareable elements and sharing mechanisms




Field sharing using a partial proxy
(*i* considered shareable)




Object sharing
(*B* considered shareable)

- Req #2 (efficiency) is not satisfied when sharing fields
- Req #4 (reflective layer) is not satisfied when sharing objects, since it breaks MOF `container()` operation

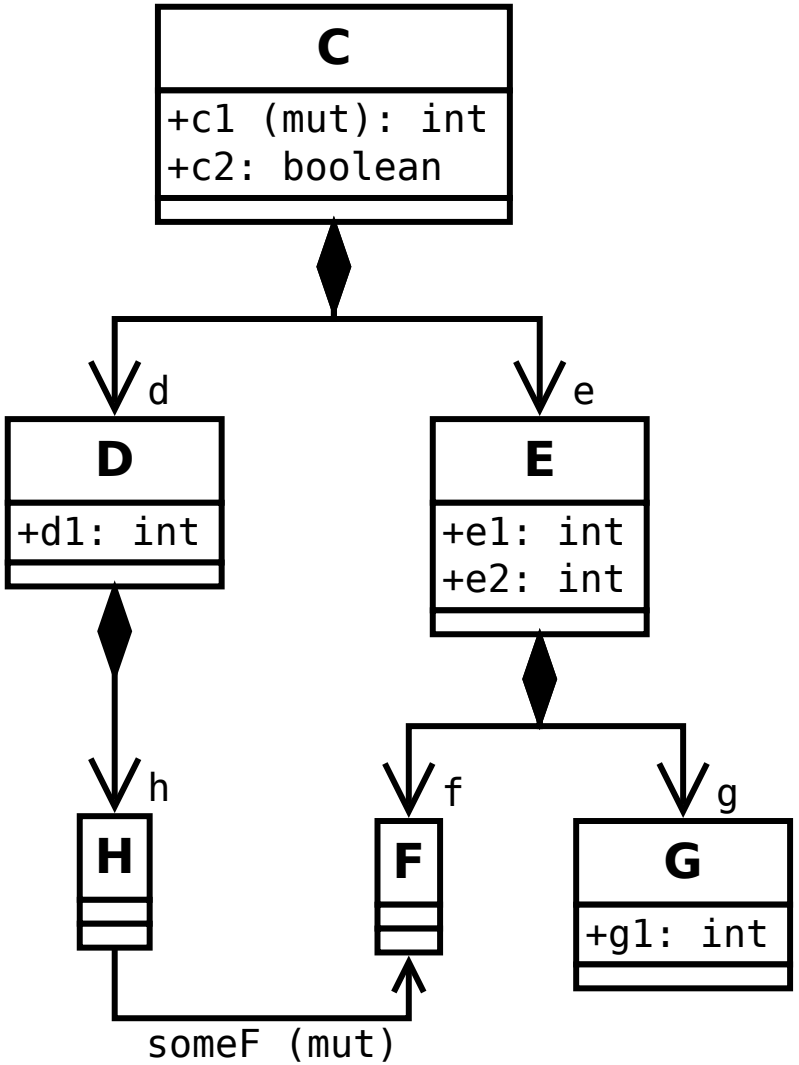
# Data sharing strategies

For design-time, **3 sharing strategies with trade-offs** between memory use and satisfaction of Req #2 and Req #4

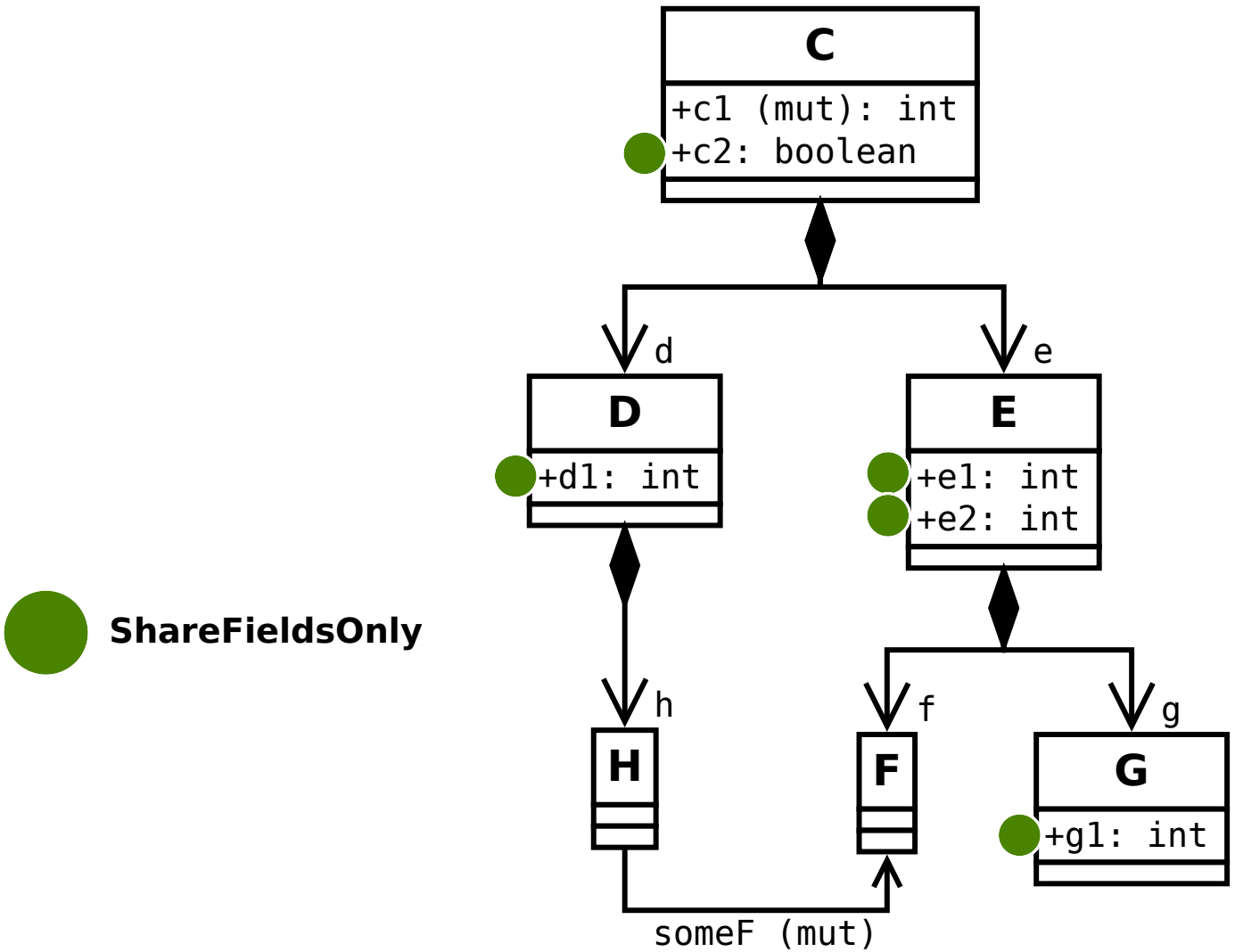
| | |
|---|---|
| **DeepCloning** | Nothing is shareable. |
| **ShareFieldsOnly** | Only immutable attributes are shareable. |
| **ShareObjOnly** | Classes that can't (transitively) access mutable parts are shareable. |
| **ShareAll** | Shareable elements are immutable attributes, classes whose properties are all shareable, and immutable references pointing to shareable classes. |

For runtime, **1 generic algorithm** parameterized by a sharing strategy $\rightarrow$ *3 data sharing cloning operators*.

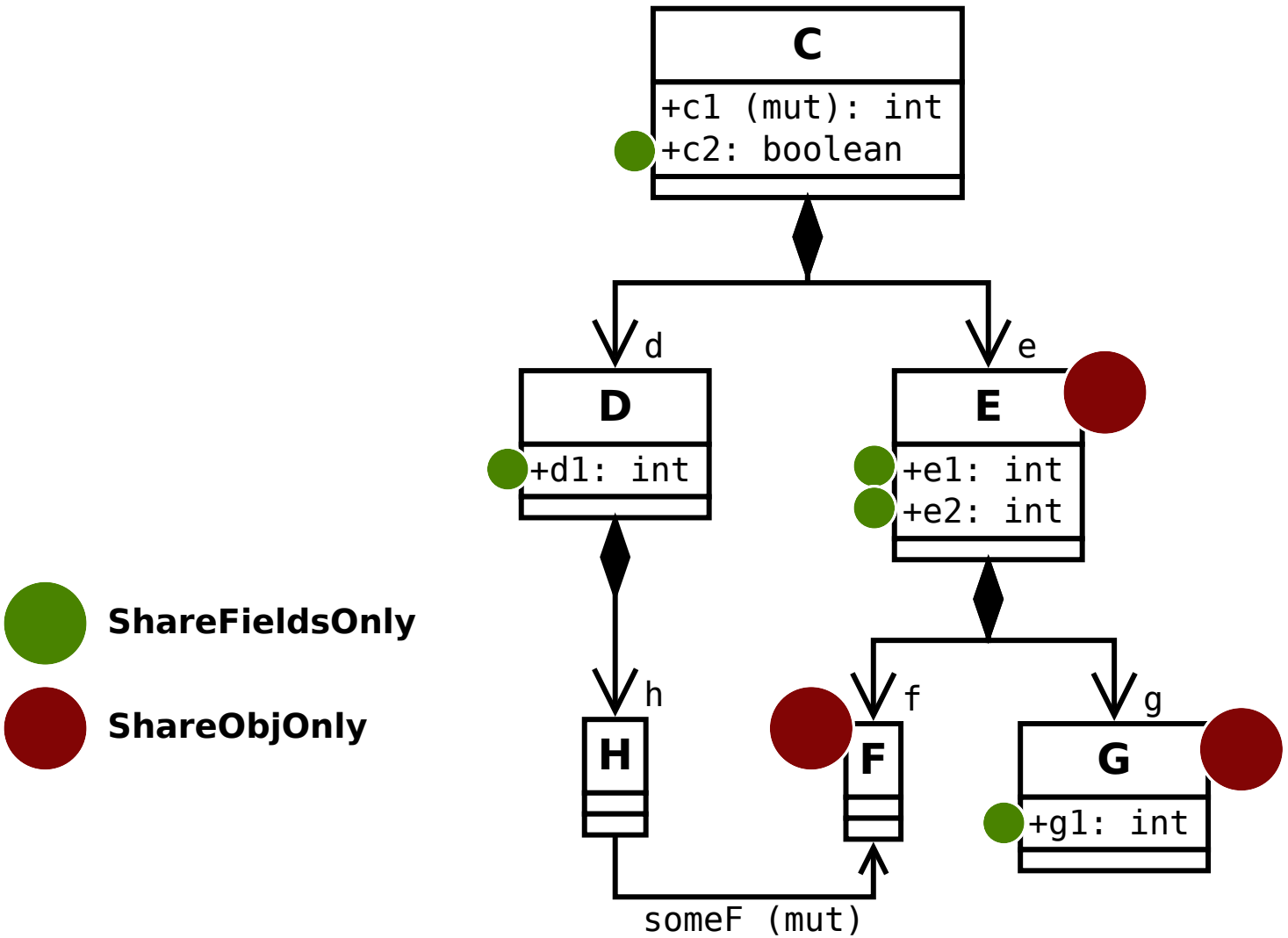
# Data sharing strategies: example



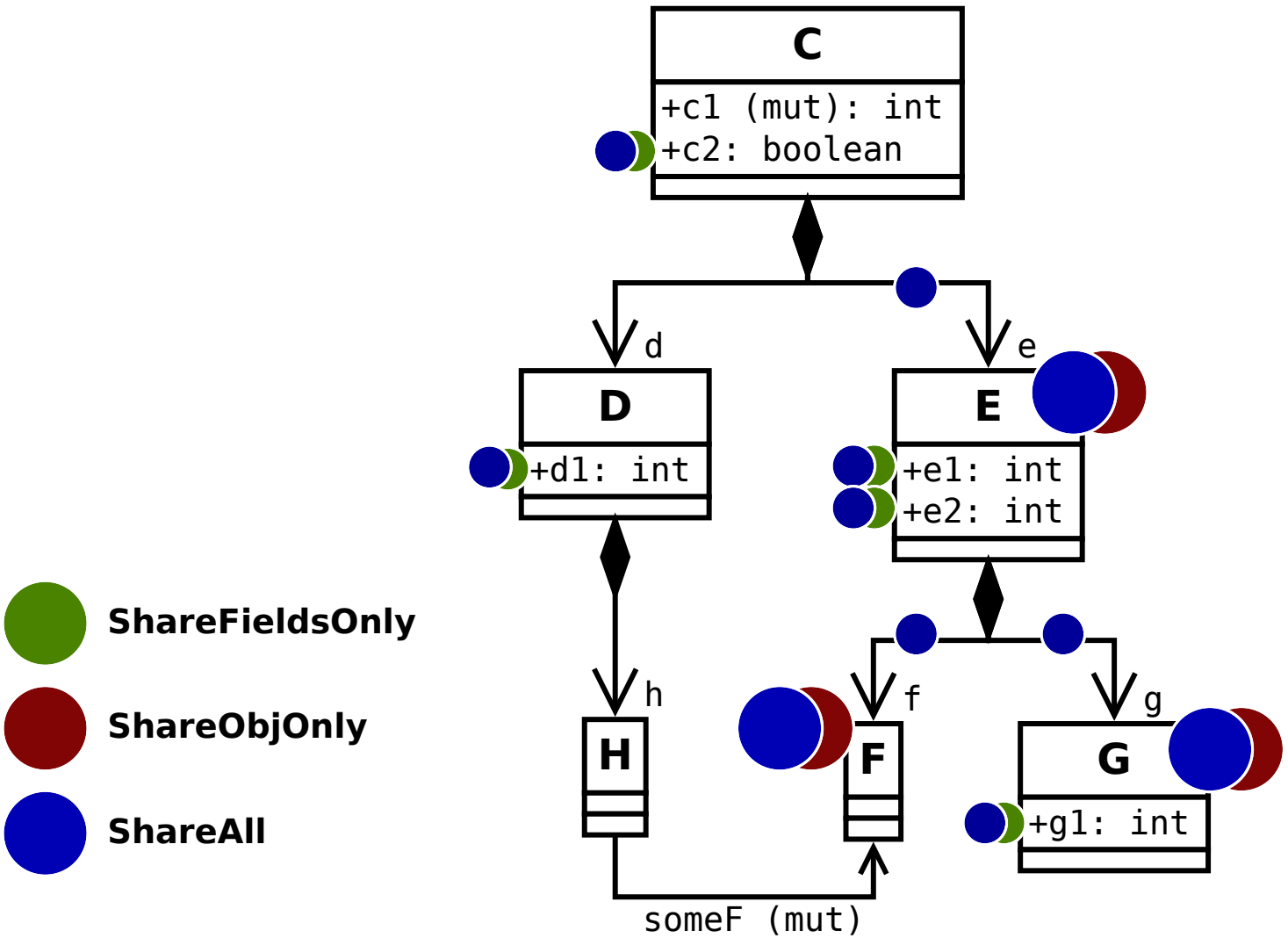
# Data sharing strategies: example



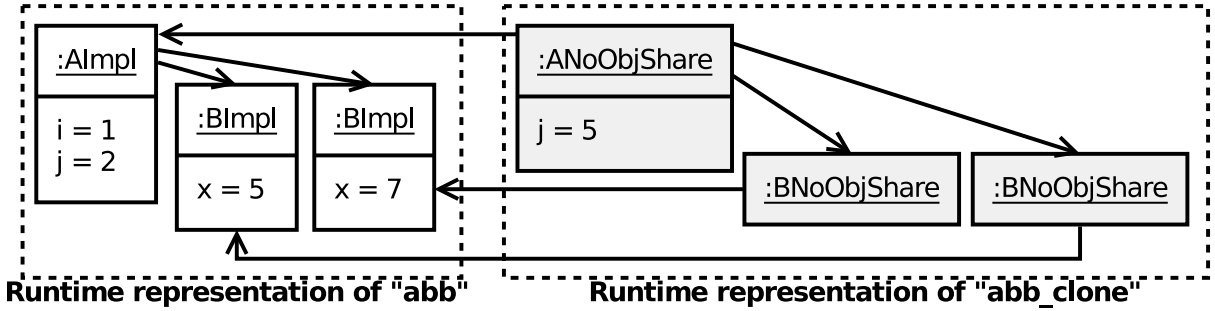
# Data sharing strategies: example



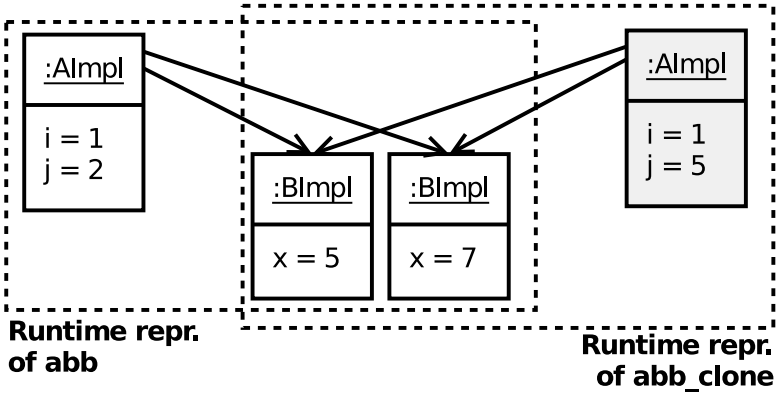
# Data sharing strategies: example



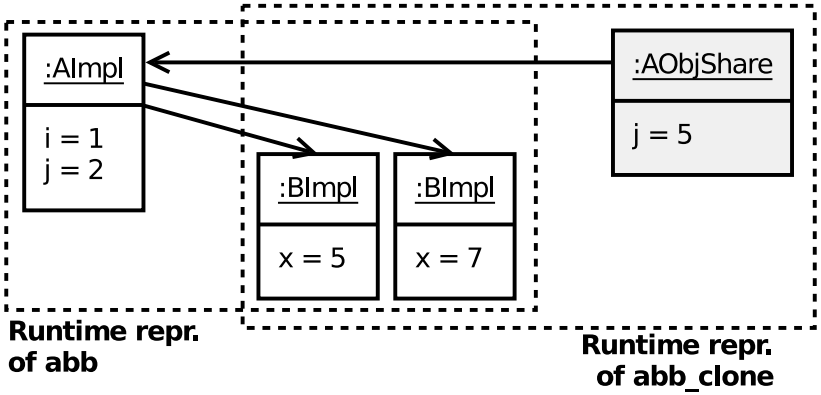
# Resulting cloning operators




(a) *ShareFieldsOnly*




(b) *ShareObjOnly*




(c) *ShareAll*

# Research questions

## RQ#1

Do the new operators reduce the memory footprint of clones, compared to deep cloning?

## RQ#2

Can a clone be manip. with the same efficiency as the original ?

## RQ#3

Can a clone be manip. using the same generated API ?

## RQ#4

Can a clone be manip. using the reflective layer (e.g. as stated in the MOF Reflection package)?

# Evaluation – RQ#1 and RQ#2

## Experiment

- **data set**: 100 randomly generated metamodels
- **memory measures**: gain as compared to *deep cloning*, after cloning the model 1000 times
- **performance measures**: loss of time as compared to the original model, when navigating 10 000 times through each object of the model while accessing all properties

## Results

- **memory**: the more shareable parts, the more memory gain
- **performance**: worst median overhead is 9,5% when manipulating clones with fields sharing

# Evaluation – results overview

RQ1: memory
RQ2: efficiency
RQ3: same API
RQ4: reflective layer

| | RQ1 | RQ2 | RQ3 | RQ4 |
|---|---|---|---|---|
| DeepCloning | ✗ | ✓ | ✓ | ✓ |
| ShareFieldsOnly | + | - - | ✓ | ✓ |
| ShareObjOnly | ++ | ✓ | ✓ | ✗ |
| ShareAll | +++ | - | ✓ | ✗ |