Trace Management Facilities to Support V&V in Executable DSLs

DiverSE SLE Seminar

Erwan Bousse[1] Benoit Combemale[2] Benoit Baudry[2]

[1]University of Rennes 1 (IRISA), France

[2]Inria, France

May 28, 2015

Outline

Plan

Context: xDSLs and traces

- Recently, a lot of effort in the field of executable Domain Specific Languages (xDSLs)
- From a Verification and Validation (V&V) point of view, need for **dynamic V&V approaches** in order to analyse the behaviors of executable models

Central concept in dynamic V&V approaches: **execution traces!**

Examples of trace usages in dynamic V&V:

- **Omniscient Debugging:** a trace is used to step backward
- **Model checking:** counter example in the form of a trace
- **Runtime monitoring:** checks if a trace satisfies a property

Context: xDSLs and traces

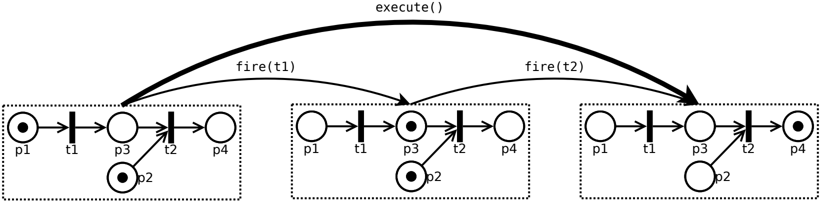
- Recently, a lot of effort in the field of executable Domain Specific Languages (xDSLs)
- From a Verification and Validation (V&V) point of view, need for **dynamic V&V approaches** in order to analyse the behaviors of executable models

Central concept in dynamic V&V approaches: **execution traces!**

Examples of trace usages in dynamic V&V:

- **Omniscient Debugging:** a trace is used to step backward
- **Model checking:** counter example in the form of a trace
- **Runtime monitoring:** checks if a trace satisfies a property

Execution Trace



Definition: Execution Trace

- An alternate sequence of *states* and *small steps*
- A *state* contains the values of all the mutable parts of a model
- A *small step* is the application of a transformation rule from the operational semantics
- A *big step* is a sequence of steps

Concerns when Managing Traces

What kind of data structure for a trace?

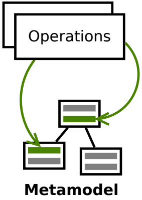
How to handle a (potentially large) trace?

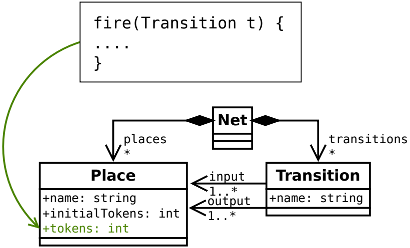
Concerns we focused on

- **Usability**, e.g. making information accessible without searching and without using type checks / casting
- **Scalability in space**, both offline (file or database storage) or online (in memory)
- **Scalability in time**, e.g. browsing a large trace

Observation: mutable subset of a metamodel

- Operational semantics only access (*read*) and change (*write*) model elements corresponding to a subset of the considered metamodel
- We call the subset concerned by *write* operations the **mutable subset of a metamodel**

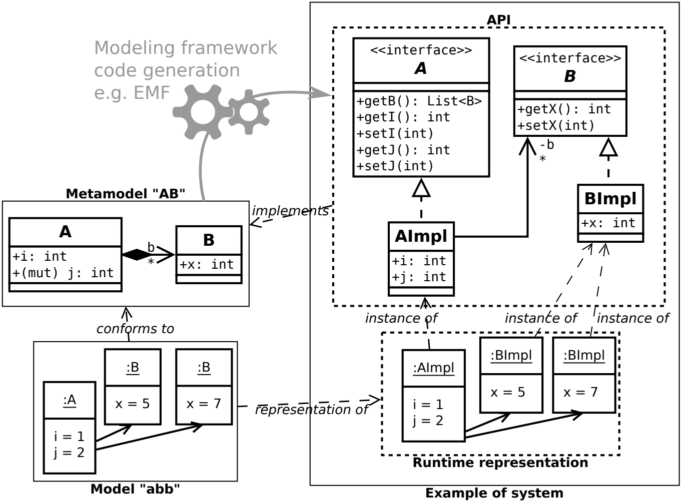




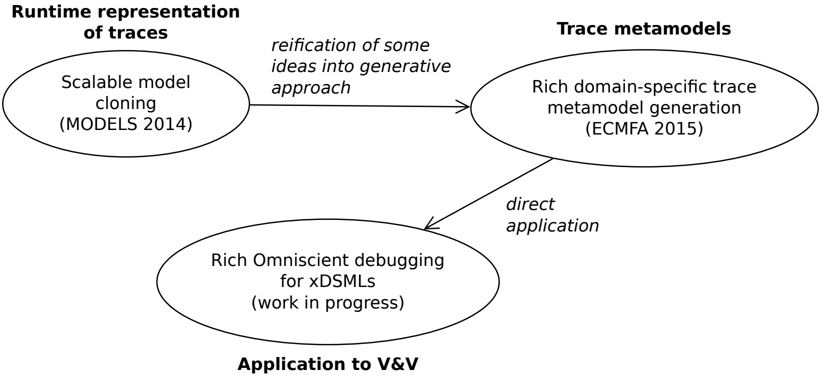
Intuition: focus on mutable subset

- When constructing the trace of an execution, we can focus on these mutable parts, and store only once the immutable parts
- In other words, we can **reduce redundancies within execution traces**
- This can be done both:
  - by working on the **runtime representation** of traces, using a memory data structure that avoids redundancy
  - by working on **traces metamodels**, in order to provide efficient representations of traces

Example: a metamodel, a model, and runtime counterparts



Overview of the contributions

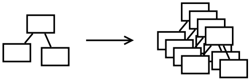


Plan

Context and motivation

Definition: clone

A *clone* is a model that is, when created, identical to an existing model. Both models conform to the same metamodel and are independent from one to another.



- New MDE activities which rely on the production of **large quantities of models and variations of a set of models**, that can be obtained through **model cloning**.
- In particular, cloning can be used to construct execution traces very conveniently

Problem: efficient model cloning

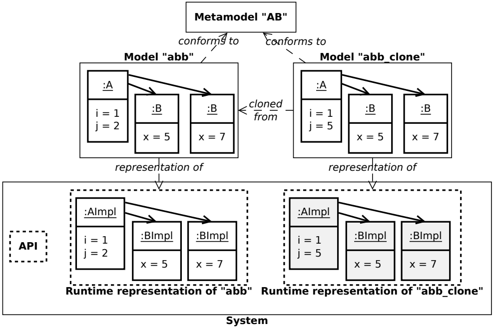
- Need for the ability to **clone a model**
- Already possible using the most convenient cloning implementation: **deep cloning** (see *EcoreUtil.Copier* class)
- *deep cloning* $\equiv$ duplicating the model in memory

Problem: *deep cloning* has very poor memory performances

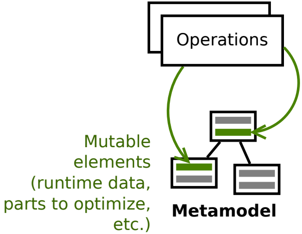
Problem: efficient model cloning

- Need for the ability to **clone a model**
- Already possible using the most convenient cloning implementation: **deep cloning** (see *EcoreUtil.Copier* class)
- *deep cloning* $\equiv$ duplicating the model in memory

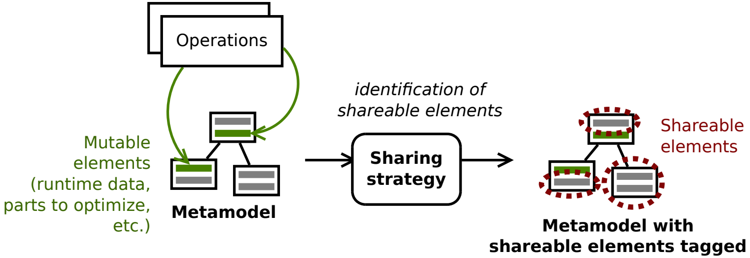
Problem: *deep cloning* has very poor memory performances



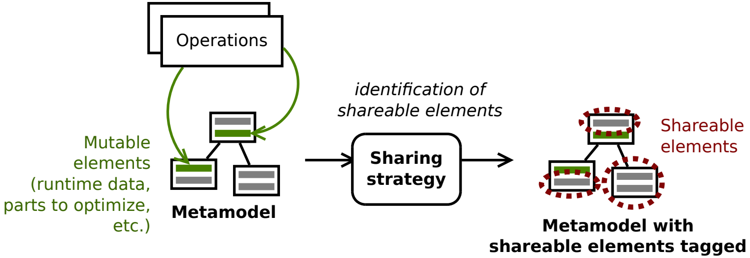
Approach: *static* identification of safely shareable parts

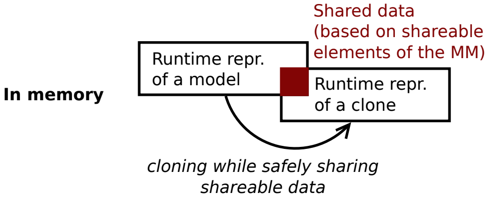


Approach: *static* identification of safely shareable parts



Approach: *static* identification of safely shareable parts





Evaluation

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

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

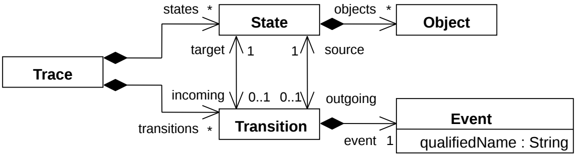
Results

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

Plan

Problem: generic trace metamodels are not good enough

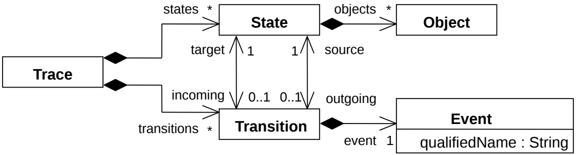
Example of Generic trace metamodel ($\equiv$ sequence of clones)



- **Efficiency issues**: sequential structure $\Rightarrow$ to navigate in a trace, each execution state has to be visited
- **Usability issues**: Domain-specific trace analyses have to handle domain-specific data that may be arbitrarily complex, and a generic set of objects is not convenient

Problem: generic trace metamodels are not good enough

Example of Generic trace metamodel ($\equiv$ sequence of clones)



- **Efficiency issues**: sequential structure $\Rightarrow$ to navigate in a trace, each execution state has to be visited
- **Usability issues**: Domain-specific trace analyses have to handle domain-specific data that may be arbitrarily complex, and a generic set of objects is not convenient

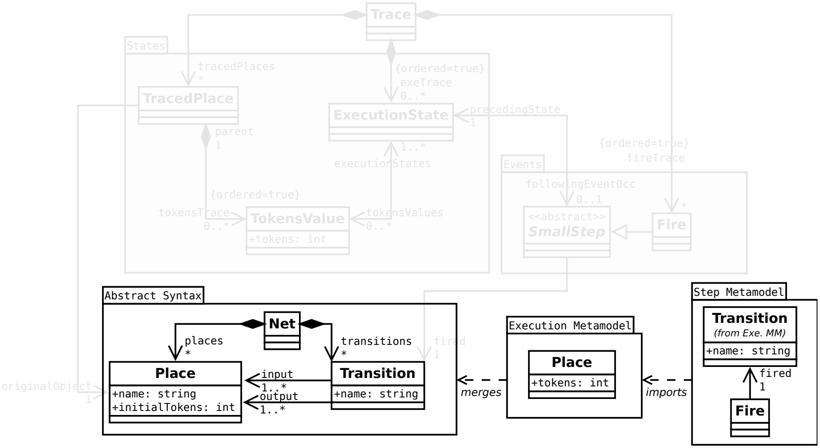
Approach: generating a domain-specific trace metamodel

1. Generative approach to automatically derive a **domain-specific trace metamodel** for a given xDSL
   - *Domain-specific*: domain concepts are directly accessible
   - *Automation*: Save language engineers the design of a complex metamodel, which is time-consuming and error-prone
2. **Enrichment** with additional navigation facilities

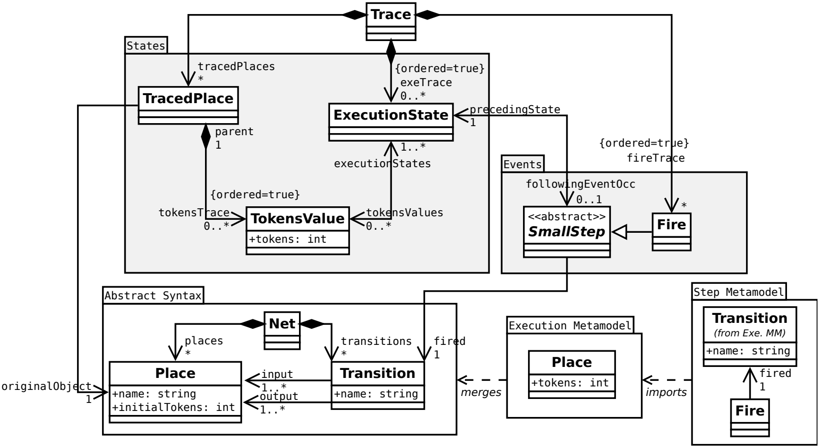
Expectations:

- Usability of traces is improved
- Efficiency of trace manipulations is improved

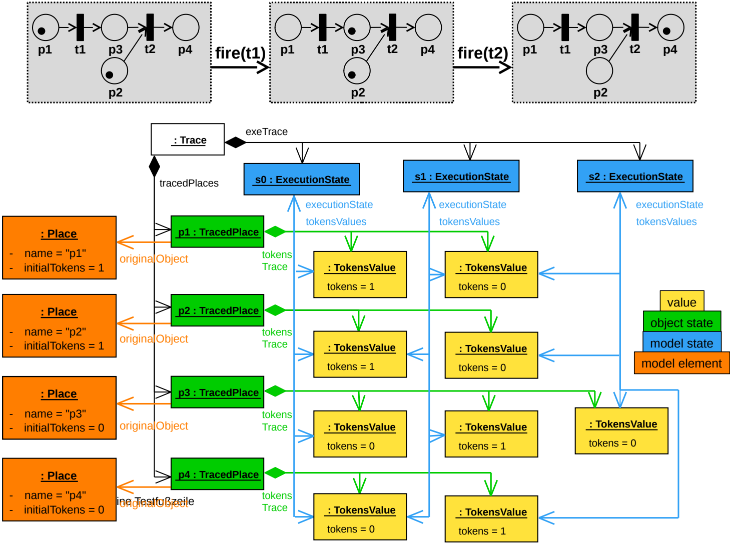
Example of trace metamodel generation



Example of trace metamodel generation



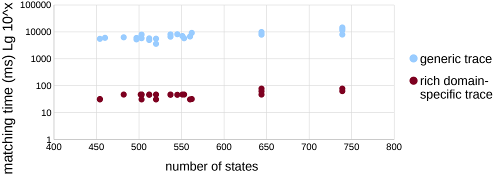
Example of Domain Specific Trace



Evaluation: performance and usability

Case study: semantic differencing (ie. trace comparison)

Efficiency in time improvement

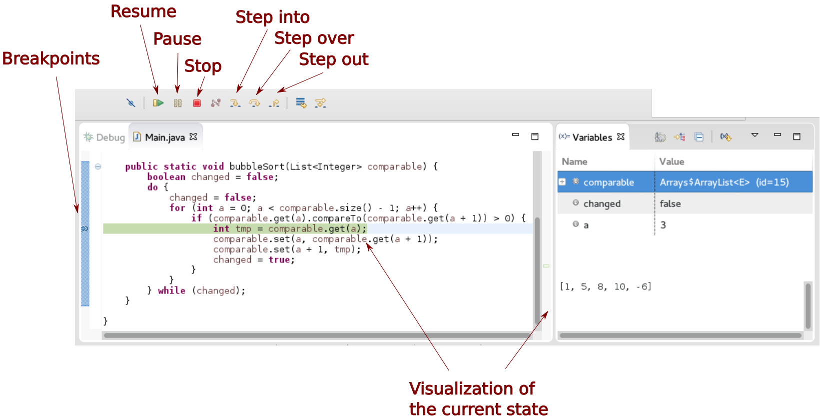


Usability improvement

Elements	Generic	Rich Domain-Specific	Gain
Lines of code	136	55	60%
Statements	58	21	64%
Operation calls	32	13	60%
Loops	5	4	40%
Type checks	4	0	100%

Plan

Traditionnal (forward-only) debugging



Also notion of **frame** (e.g. a method in Java) into which we can *step* or from which we can *step out*.

Omniscient debugging

- A debugger only gives services to go **forward in time**
- Omniscient debugging aims at providing operators that can go **backwards in time** as well
  - **jump**: to go back to a chosen previous state
  - **step back into**: go back one step
  - **step back over**: go back one step over a frame
  - **step back out**: go back out of a frame
  - **play backwards**: goes back though all previous states
  - **visualization of history**: a representation of the previous states (counter, trace, etc.)

Omniscient debuggers usually rely on **an execution trace** to store previous states and implement these services.

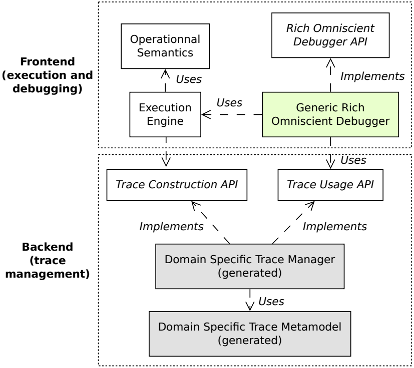
Omniscient debugging

- A debugger only gives services to go **forward in time**
- Omniscient debugging aims at providing operators that can go **backwards in time** as well
  - **jump**: to go back to a chosen previous state
  - **step back into**: go back one step
  - **step back over**: go back one step over a frame
  - **step back out**: go back out of a frame
  - **play backwards**: goes back though all previous states
  - **visualization of history**: a representation of the previous states (counter, trace, etc.)

Omniscient debuggers usually rely on **an execution trace** to store previous states and implement these services.

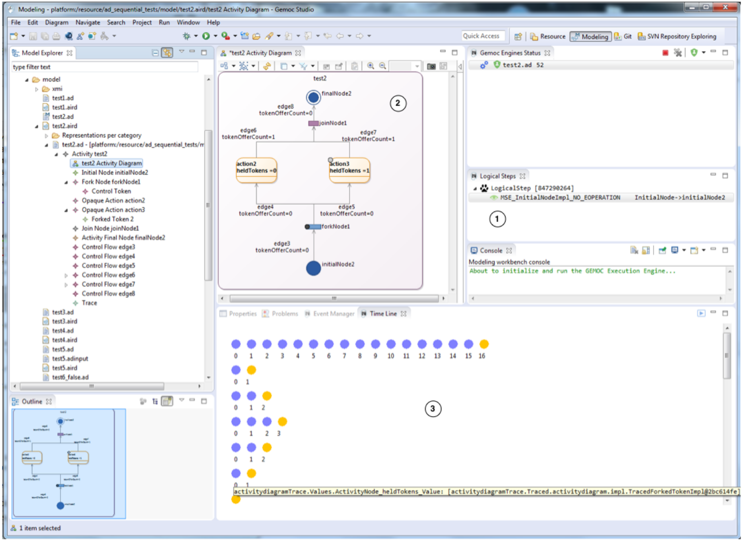
Overview of the contribution (work in progress)

A generative approach for **rich omniscient debugging** for xDSLs



- rich because of **new jump services**, e.g. jump to a specific field value
- allows the reuse of domain-specific traces in other domain-specific trace activities

Implementation prototype in the GEMOC Studio



Plan

Conclusion

- xDSLs brings a lot of possibilities: simulation, dynamic V&V
- Dynamic V&V requires execution traces, which can be large and difficult to store/use
- Two studied approaches:
  - at the **runtime representation level**: Scalable Model Cloning
  - at the **trace metamodel level**: Rich domain-specific Trace metamodel generation
- Application to rich omniscient debugging for executable DSLs

Perspectives

- Enhancing customisation of trace metamodels
- Experiment other V&V activities on top of generated trace metamodels

Done!

Thank you for your attention! ☺