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)
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

- 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

- $\nabla$-Nabla (Numerical-analysis)
- CATIA (Computed-aided manufacturing)
- HTML (Web development)
- POV-Ray (Computer graphics)
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

- Semantics
- Abstract Syntax
- Concrete Syntax

DSL
Anatomy and tooling of a DSL
Anatomy and tooling of a DSL

- Abstract Syntax
- Concrete Syntax
- Semantics

- Editor
- Debugging
- Refactoring
- Static Checker
- Tracer
- Runtime monitoring
- Dynamic Checker
- Test runner
- Documentation generator
- Etc.
Example of (executable) DSL
Example of (executable) DSL

Petri net model
Example of (executable) DSL

Abstract Syntax

Net

places

transitions

Place
+name: string
+initialTokens: int

Transition
+name: string

transitions

Execution Metamodel

Place
+tokens: int

Operational semantics

Conforms to

run(Net) : while there is an enabled transition, fires it.
firg(Transition) : removes a token from each input Place, and adds a token to each output Place.

Execution transformation rules (summarized)

Petri net model
Example of (executable) DSL

Abstract Syntax

- **Net**
  - **Place**
    - name: string
    - initialTokens: int
  - **Transition**
    - name: string

- transitions

- places

Run(Net)
  : while there is an enabled transition, fire it.

Fire(Transition)
  : removes a token from each input Place, and adds a token to each output Place.

Execution transformation rules (summarized)

Petri net model

- init=1
- p1
- init=0
- t1
- init=0
- p3
- t2
- p4

Execution model

- init=1
- p1
- t1
- p3
- t2
- p4

Initial state

1. Petri net model
2. Execution model
3. Operational semantics
4. Conforms to

8 / 41
Example of (executable) DSL

Abstract Syntax

```
Net
+places (* *)
+transitions (* *)
```

```
Place
+name: string
+initialTokens: int
```

```
Transition
+name: string
```

Operational semantics

Execution Metamodel

```
Place
+tokens: int
```

Execution transformation rules (summarized)

- `run(Net)`
  - while there is an enabled transition, fires it.
- `fire(Transition)`
  - removes a token from each input Place, and adds a token to each output Place.

Petri net model

```
init=1
p1
  \[\rightarrow\]
  \[\rightarrow\]
init=0
p2
  \[\rightarrow\]
init=0
p3
init=0
p4
```

```
init=1
p2
  \[\rightarrow\]
  \[\rightarrow\]
```

```
\[\rightarrow\]
\[\rightarrow\]
```

Petri net model

```
initialization
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
```

```
\[\rightarrow\]
Example of (executable) DSL

Abstract Syntax

Net

places

transitions

Place

+ name: string
+ initialTokens: int

Transition

+ name: string
+ input 1..*
+ output 1..*

Execution Metamodel

Place

+ tokens: int

Operational semantics

run(Net)
fire(Transition)

: while there is an enabled transition, fires it.
: removes a token from each input Place, and adds a token to each output Place.

Execution transformation rules (summarized)

Petri net model

init=1
p1
init=0
p2
init=0
t1
p3
t2
init=1
p4

Initial model

Execution model

(t1 fired) (t2 fired)
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*!
Instead of restricting a tool to a given DSL, can we make tools compatible with a *wide range of DSLs*?

In other words: can we make truly *generic* tools?
Idea

- Instead of restricting a tool to a given DSL, can we make tools compatible with a *wide range of DSLs*?
- 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 (i.e. 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 (e.g. "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

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?
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 (e.g., 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 (e.g., 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 (e.g. 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?
Enrichment of the DSL with steps and states

**Abstract Syntax**

- **Net**
  - places: \( \ast \)
  - transitions: \( \ast \)

- **Place**
  - name: string
  - initialTokens: int

- **Transition**
  - input: \( 1..* \)
  - output: \( 1..* \)

**Execution Metamodel**

- **Place**
  - tokens: int

**Execution transformation rules (summarized)**

- **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
Enrichment of the DSL with steps and states

Abstract Syntax

Net
- places
  - input
    - transitions
  - output
- transitions

Place
- name: string
  - initialTokens: int

Transition
- name: string
  - input
    - 1..*
  - output
    - 1..*

Execution Metamodel

Place
- tokens: int

Enrichment

@Step run(Net): while there is an enabled transition, fires it.
@Step 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.
@Step addToken(Place): adds a token to a Place
@Step removeToken(Place): removes a token from a Place

Execution transformation rules (summarized)
2 – Generic generation of a DSL-specific tracer

![Diagram showing the relationship between DSL, execution of a model, and tracing tool generator.]

- **DSL**
  - Semantics
  - Syntax

- **Execution of a model**
  - `run()`
  - `fire(t1)`
  - `fire(t2)`

- **Tracing Tool Generator**

- **Tracer**
  - Trace metamodel
  - Trace constructor

- **Observe**

- **Conforms to**

- **Specific to**
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
   ```java
   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**
1. Reification of mutable properties 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
1. Reification of mutable properties 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.
1. Reification of mutable properties 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 properties 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 properties 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
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

1 begin
2   changes ← getFieldChanges()
3   state_current ← root.executionStates.last()
4   if state_current = null then
5     addInitialState(root, model_exe, map_traced)
6   else if changes ≠ ∅ then
7     state_new ← copyState(state_current)
8     root.executionStates.add(newState)
9     foreach fieldChange ∈ changes do
10        o ← fieldChange.changedObject
11        traced_o ← map_traced(o)
12        if o.is(Place) then
13           p ← fieldChange.changedProperty
14           if p.is(Place.tokens) then
15              v_new ← createObject(TokensValue)
16              v_new.tokens = o.tokens
17              v_old ← traced_o.tokensSequence.last()
18              state_new.tokensValues.remove(v_old)
19              state_new.tokensValues.add(v_new)
20              traced_o.tokensSequence.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

1 begin
2   step_new ← null
3   state_current ← root.executionStates.last()
4   step_current ← stack_steps.peek()
5   if stepRuleID = getRuleID(RunStep) then
6     step_new ← createObject(RunStep)
7     step_new.caller ← stepRuleParams[0]
8     root.runSequence.add(step_new)
9     root.rootSteps.add(step_new)
10    else if stepRuleID = getRuleID(FireStep) then
11       step_new ← createObject(FireStep)
12       step_new.caller ← stepRuleParams[0]
13       root.fireSequence.add(step_new)
14       _step_current.subSteps.add(step_new)
15       step_new.startingState ← state_current
16       stack_steps.push(step_new)
Example of Petri net trace (states only)
Case 2: Generic 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)
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 (e.g. Petri nets)

We must re-think interactive debugging in a language-agnostic fashion
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 (e.g. "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.
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

Omniscient debugger

Execution of a model

conforms to

configured by

observe, control

debbuging services

DSL

Semantics

Syntax

consists of

Execution of a model
Generic omniscient debugging metamodel

<<abstract>>
Breakpoint
stepping: boolean

<<enum>>
StepNotificationKind
STARTING
ENDING

<<abstract>>
DebuggingState
lastNotification: StepNotificationKind
exeState
1

<<abstract>>
ExecutionState
stepping
1

<<abstract>>
Stepping

<<abstract>>
ModelState
modelState
1

<<abstract>>
Step

{ordered=true}
inProgress

<<abstract>>
Generic Trace
ending
0..1

starting
0..1

breakpoints
*

Generic omniscient debugging metamodel

Trace metamodel – States concepts

Generic omniscient debugging metamodel

29 / 41
### Generic definition of debugging services (excerpt)

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
</table>
| **stepInto** | [1] if `dstate.exeState_stepping_starting` ≠ null then  
|              | [2] `dstate.breakpoints.add([true])`  
|              | [3] `_play()`                                                                          |
| **stepOver** | [1] if `dstate.exeState_stepping_starting` ≠ null then  
|              | [2] `s_over ← dstate.exeState_stepping_starting`  
|              | `dstate.breakpoints.add([dstate.exeState_stepping_ending = s_over])`  
|              | [3] `_play()`                                                                          |
| **stepOut**  | [1] if `dstate.exeState_steppingInProgress` ≠ ∅ then  
|              | [2] `s_out ← dstate.exeState_steppingInProgress.peak()`  
|              | `dstate.breakpoints.add([dstate.exeState_stepping_ending = s_out])`  
|              | [3] `_play()`                                                                          |

Table 3: Forward services of the omniscient debugger
Example of Petri net generic omniscient debugging
Connecting generic tools to DSLs at runtime

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
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
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
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
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 it 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?
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)*?
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

Thank you for your attention!

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