Domain-Level Observation and Control for Compiled Executable DSLs

MODELS 2019 Foundations Track – Münich, Germany

Erwan Bousse
University of Nantes – LS2N, France

Manuel Wimmer
CDL-MINT, Johannes Kepler University Linz, Austria
Behavioral models (eg. state machines) can conveniently describe the behaviors of systems under design.

Domain-specific languages (DSLs) can be engineered and used to build such models.

Dynamic analyses of behavioral models are crucial in early design phases to see how a described behavior unfolds over time.

Require the possibility to execute models.
Behavioral models (eg. state machines) can conveniently describe the behaviors of systems under design.

Domain-specific languages (DSLs) can be engineered and used to build such models.

Dynamic analyses of behavioral models are crucial in early design phases to see how a described behavior unfolds over time.

Require the possibility to execute models!
Behavioral models (eg. state machines) can conveniently describe the behaviors of systems under design.

Domain-specific languages (DSLs) can be engineered and used to build such models.

Dynamic analyses of behavioral models are crucial in early design phases to see how a described behavior unfolds over time.

Require the possibility to execute models 🌐!
Behavioral models (eg. state machines) can conveniently describe the behaviors of systems under design.

Domain-specific languages (DSLs) can be engineered and used to build such models.

Dynamic analyses of behavioral models are crucial in early design phases to see how a described behavior unfolds over time.

Require the possibility to execute models!
Model execution with an interpreted DSL
Model execution with an interpreted DSL

- Dependency
- Conforms to
- Input/output

Interpreted DSL
Model execution with an interpreted DSL

Interpreted DSL

Abstract Syntax (metamodel)

- dependency
- conforms to
- input/output

Model

Procedure
Model execution with an interpreted DSL
Model execution with an interpreted DSL
Model execution with an interpreted DSL
Model execution with an interpreted DSL
Model execution with an interpreted DSL
Model execution with an interpreted DSL
Example of an Interpreted DSL

Abstract Syntax

- Net
  - places
  - transitions

- Place
  - name: String
  - initialTokens: Integer

- Transition
  - name: String
  - input
  - output

Model State Definition

- Place
  - tokens: Integer

- Transition
  - tokens: Integer

Interpretation rules (summarized)

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

Net

places

transitions

Place
name: String
initialTokens: Integer

Transition
name: String

Model State Definition

Place
tokens: Integer

Interpretation rules (summarized)

run(Net) fire(Transition)

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

Example of an Interpreted DSL
Example of an Interpreted DSL

Abstract Syntax

Net
- places
- transitions

Place
- name: String
- initialTokens: Integer

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

Model State Definition

Place
- tokens: Integer

Transition
- name: String

Net
- places
- transitions

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

Interpretation rules (summarized)

A model state
- foo(): step
- observation point

1..*
Example of an Interpreted DSL

Abstract Syntax

Net

name: String
initialTokens: Integer

Place

Transitions

input 1..*
output 1..*

Model State Definition

Place

tokens: Integer

Transition

name: String

Interpretation rules (summarized)

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

Dynamic Analysis

Dynamic Analysis

A

1

observation point

step

foo() model state

B

C

D

1

2

3

4

5

6
Debugging/Tracing an interpreted model in the GEMOC Studio
Question

What about DSLs built with a compiler (eg. a code generator) instead of an interpreter?
Model execution with a compiled DSL
Model execution with a compiled DSL
Model execution with a compiled DSL
Model execution with a compiled DSL
Model execution with a compiled DSL

Compiled DSL

Source Abstract Syntax

Compiled DSL

Compiler

Target Language (interpreted)

Target Abstract Syntax

Target Model State Definition

Target Execution Steps Definition

Interpretation Rules

Target Engine

Model

dependency

conforms to

input/output

Procedure
Model execution with a compiled DSL
Model execution with a compiled DSL
Model execution with a compiled DSL
Example of a compiled DSL (1)

AD abstract syntax

```
<<abstract>>
NamedElement
+name: String
```

```
<<abstract>>
Node
```

```
Activity
```

```
Edge
```

```
InitialNode  Action  ForkNode  JoinNode  FinalNode
```

```
transformActivity(Activity) : Creates a Net
transformEdge(Edge) : Creates a Place
transformAction(Action) : Creates a Place and two Transitions
...
Example of a compiled DSL (2)

Source activity diagram
Example of a compiled DSL (2)

Source activity diagram

Target Petri net obtained after compilation
Problem: Dynamic analysis is performed at the level of the target domain!
ie. when debugging activity diagrams, we must use a petri nets debugger:
The case of programming languages
Most **general-purpose programming languages** rely on efficient compilers for their semantics, either targeting some form of bytecode (eg. Java or Python) or machine code (eg. C or C++).
Most general-purpose programming languages rely on efficient compilers for their semantics, either targeting some form of bytecode (e.g., Java or Python) or machine code (e.g., C or C++).

Most of these languages do provide an interactive debugger at the source domain level to step through the execution and observe the program state.
Most general-purpose programming languages rely on efficient compilers for their semantics, either targeting some form of bytecode (eg. Java or Python) or machine code (eg. C or C++).

Most of these languages do provide an interactive debugger at the source domain level to step through the execution and observe the program state.

But these debuggers result from ad-hoc language engineering work! This does not give us a systematic recipe for engineering new DSLs.
Most general-purpose programming languages rely on efficient compilers for their semantics, either targeting some form of bytecode (e.g., Java or Python) or machine code (e.g., C or C++).

Most of these languages do provide an interactive debugger at the source domain level to step through the execution and observe the program state.

But these debuggers result from ad-hoc language engineering work! This does not give us a systematic recipe for engineering new DSLs.

How can we engineer compiled DSLs compatible with dynamic analyses at the source domain level, just as common general-purpose programming languages?
Contribution

An architecture to support observation and control for compiled DSLs.
Approach Overview
Approach Overview (1)
Approach Overview (2)
Step (b) – Source model state definition
Observing the execution of a model requires accessing its state as it changes (tokens, variables, activated elements, etc.).
Observing the execution of a model requires accessing its state as it changes (tokens, variables, activated elements, etc.).

For interpreted DSLs, possible states are defined by a model state definition which extends the abstract syntax of the DSL with new dynamic properties and metaclasses (eg. tokens for the Petri nets DSL).
Observing the execution of a model requires accessing its state as it changes (tokens, variables, activated elements, etc.).

- For interpreted DSLs, possible states are defined by a model state definition which extends the abstract syntax of the DSL with new dynamic properties and metaclasses (eg. tokens for the Petri nets DSL).
- But for compiled DSLs, everything related to execution is delegated to the target language, including the state definition.
Observing the execution of a model requires accessing its state as it changes (tokens, variables, activated elements, etc.).

- For interpreted DSLs, possible states are defined by a model state definition which extends the abstract syntax of the DSL with new dynamic properties and metaclasses (eg. tokens for the Petri nets DSL).

- But for compiled DSLs, everything related to execution is delegated to the target language, including the state definition.

- Hence, necessary to extend a compiled DSL with a model state definition, to define explicitly the possible states of conforming source models.
When executing a UML activity diagram, tokens flow through both nodes and edges of the model.

We add a TokensHolder metaclass to reflect that:
Approach Overview (2)
Approach Overview (3)
Step (c) – Source execution steps definition
Observing and controlling require knowing the **execution steps** of the model execution, ie. what are the **observable changes** made to the state.
Observing and controlling require knowing the **execution steps** of the model execution, i.e. what are the **observable changes** made to the state.

For **interpreted DSLs**, specific interpretation rules can be tagged as producers of execution steps (e.g. the **fire** step for Petri nets).
Observing and controlling require knowing the **execution steps** of the model execution, i.e. what are the **observable changes** made to the state.

- For **interpreted DSLs**, specific interpretation rules can be tagged as producers of execution steps (e.g. the **fire** step for Petri nets).
- For **compiled DSLs**, we propose a trivial **step definition metamodel** to declare possible execution steps.
Observing and controlling require knowing the execution steps of the model execution, i.e. what are the observable changes made to the state.

- For interpreted DSLs, specific interpretation rules can be tagged as producers of execution steps (e.g. the fire step for Petri nets).
- For compiled DSLs, we propose a trivial step definition metamodel to declare possible execution steps.
Example of execution steps definition for the AD DSL
In UML activity diagrams, a node will **take** tokens from incoming edges, and **offer** tokens on its outgoing edges when it finishes its task.
In UML activity diagrams, a node will take tokens from incoming edges, and offer tokens on its outgoing edges when it finishes its task.

We define the following execution steps to reflect that:

- offer(Node): offering of tokens of a Node to the outgoing edges of the Node;
- take(Node): taking of tokens by a Node from the incoming edges of the Node;
- executeNode(Node): taking and offering of tokens by a Node, i.e., a composite step containing both an offer step and a take step;
- executeActivity(Activity): execution of the Activity until no tokens can be offered or taken, i.e., a composite step containing executeNode steps.
Approach Overview (4)
Now remains the translation at runtime of states and steps of the target model back to the source model, to be observed by dynamic analysis tools.
Now remains the translation at runtime of states and steps of the target model back to the source model, to be observed by dynamic analysis tools.

Our approach: definition of a feedback manager attached to the execution, which performs said translation on the fly during the model execution.
Now remains the translation at runtime of states and steps of the target model back to the source model, to be observed by dynamic analysis tools.

**Our approach**: definition of a feedback manager attached to the execution, which performs said translation on the fly during the model execution.

Proposed interface for feedback managers:

- `feedbackState`: Update the source model state based on the set of changes applied on the target model state in the last target execution step.
- `processTargetStepStart`: Translate a target starting step into source steps.
- `processTargetStepEnd`: Translate a target ending step into source steps.
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
Example of execution reconstructed by a feedback manager

Target Petri net execution trace (invisible to users and tools)

Source activity diagram execution trace (seen by users and tools)
 Approach Overview (4)
Approach Overview (5)
Evaluation

Can we observe and control compiled models?
In reasonable time?
Common parts (eg. glue code, APIs, integration layer) of the approach implemented for the **GEMOC Studio**, an Eclipse-based language workbench.

- The source code (Eclipse plugins written in Xtend and Java) is available on Github: https://github.com/tetrabox/gemoc-compilation-engine

**Note**

As he GEMOC Studio originally focused on interpreted DSLs, *this is the first attempt to support compiled DSLs in the GEMOC Studio.*
Evaluation: RQs
RQ#1

Given an interpreted DSL and a compiled DSL with trace-equivalent semantics, does the approach make it possible to observe the same traces with both DSLs?
**RQ#1**
Given an interpreted DSL and a compiled DSL with trace-equivalent semantics, does the approach **make it possible to observe the same traces** with both DSLs?

**RQ#2**
Does the approach **enable the use of runtime services at the domain-level** of compiled DSLs?
RQ#1
Given an interpreted DSL and a compiled DSL with trace-equivalent semantics, does the approach make it possible to observe the same traces with both DSLs?

RQ#2
Does the approach enable the use of runtime services at the domain-level of compiled DSLs?

RQ#3
What is the time overhead when executing compiled models with feedback management?
Evaluation: Setup
Evaluation: Setup

Considered DSLs – 2 UML-based languages

- a subset of fUML activity diagrams, using Petri nets as a target language,
- a subset of UML state machines using a subset of Java as a target language.

*Each DSL implemented twice: one interpreted variant and one compiled variant.*
Evaluation: Setup

Considered DSLs – 2 UML-based languages

- a subset of fUML activity diagrams, using Petri nets as a target language,
- a subset of UML state machines using a subset of Java as a target language.

Each DSL implemented twice: one interpreted variant and one compiled variant.

Considered Runtime Services – 2 tools from our previous work

- a trace constructor (ECMFA 2015, SoSym 2017)
- an omniscient debugger (SLE 2015, JSS 2018)
Evaluation: Setup

Considered DSLs – 2 UML-based languages

- a subset of fUML activity diagrams, using Petri nets as a target language,
- a subset of UML state machines using a subset of Java as a target language.

Each DSL implemented twice: one interpreted variant and one compiled variant.

Considered Runtime Services – 2 tools from our previous work

- a trace constructor (ECMFA 2015, SoSym 2017)
- an omniscient debugger (SLE 2015, JSS 2018)

Considered Models – random generation

- 100 fUML activity diagrams in 10 groups ranging from 10 to 100 nodes,
- 30 UML state machines from 10 to 100 states, and 3 scenarios per state machine.
Evaluation: Results
RQ#1: same traces between interpreted and compiled variants?

- all 130 generated models executed with the interpreted and the compiled variants of both executable DSLs
- no difference found found when comparing traces
**RQ#1: same traces between interpreted and compiled variants?**

- all 130 generated models executed with the interpreted and the compiled variants of both executable DSLs
- **no difference found** found when comparing traces

**RQ#2: working runtime services?**

- both runtime services (trace constructor and omniscient debugger) **work as expected at the domain-level**
RQ#1: same traces between interpreted and compiled variants?

- all 130 generated models executed with the interpreted and the compiled variants of both executable DSLs
- no difference found found when comparing traces

RQ#2: working runtime services?

- both runtime services (trace constructor and omniscient debugger) work as expected at the domain-level

RQ#3: execution time overhead when using the feedback manager?

- fUML activity diagrams → Petri nets: 1,6 times slower on average
- UML State Machines → Minijava: 1,01 times slower on average
Summary

- Observing and controlling the execution of compiled models is difficult, and there is a lack of systematic approach to design compiled DSLs with that goal in mind.
- Our proposal: a generic language engineering architecture to define explicit feedback management in compiled DSLs.

Perspectives (excerpt)

- handling compilers defined as code generators;
- provide an easier way to define feedback managers;
- managing stimuli sent to the source model during the execution;
- measuring the amount of effort required to define a feedback manager as compared to defining an interpreter.
Thank you!

Github: https://github.com/tetrabox/gemoc-compilation-engine

Twitter: @erwan_bousse

Email: erwan.bousse@ls2n.fr