Scenario tutorial day two: Part 3: Hands-on GEMOC, the logo turtle language

This document indicates the steps to follow in order to reproduce the tutorial.

Prerequisite

This part follows the work done in part2. While you can start from your own work, I suggest to replace it and start from the solution of part2.

so the starting projects are available in the folder part2-mmfirst-solution . (or download the zip and use import project)

The solution of this tutorial is available in the folder part3-mmfirst-solution .

General principles

We'll create an execution semantic for the behavior or our Logo language.

It will provide features such as:

- debugger
- animation

NOTE	In order to keep the tutorial as simple as possible, we consider that we can modify the metamodel directly. This is correct for most languages you create by your own, but in some situation you may prefer to keep a stricter separation between the static and dynamic parts of the language. GEMOC offers several strategies to achieve this but they are out of the scope of this tutorial.

Define the dynamic semantic

Create a GEMOC Sequential project

- *File → New → GEMOC Sequential XDSML Project*
  - name: fr.inria.sed.logo.xdsml
- *Next*
  - use the default template: Simple Sequential K3 language project
  - Ecore file location: select the ecore file in *fr.inria.sed.logo.model/model/Logo.ecore*
- *Finish*

Create the project for domain specific action (DSA):

- right click on the *fr.inria.sed.logo.xdsml* project → GEMOC Language → create DSA project for language
- *next*
  - Select template: *User Ecore Basic Aspects*
- *next*
  - Aspect package prefix: fr.inria.sed.logo.k3dsa
  - Aspect package suffix: .aspects
  - Aspect file name: LogoAspects
- *Finish*
- add missing dependency from project *fr.inria.sed.logo.xdsml* to *fr.inria.sed.logo.k3dsa*. *(This action can be removed when modeldebugging bug #51 is fixed).*

Write a GEMOC based method to "hello world"

In the k3dsa project,

add a plugin dependency to `org.eclipse.gemoc.commons.eclipse.messagingsystem.api`

open the logoAspects.xtend file.

add the following imports:

```
import fr.inria.diverse.k3.al.annotationprocessor.Main
```

Add a *run* method with **@Main** annotation in the class LogoProgramAspect.

```
    @Main
    def void run(){
        // println('hello world')
        val MessagingSystemManager msManager = new MessagingSystemManager
        val ms = msManager.createBestPlatformMessagingSystem("Logo","Simple Logo
interpreter")

        ms.debug("Hello world on "+_self.eResource.URI, "Logo")
    }
```

Launch the **Modeling workbench**.

- *Run → Debug configurations...*
  - Right click on *Gemoc Sequential eXecutable Model → new configuration*
    - Name: <your model file name>

        - model to execute: browse and select the model file
        - Languages: _fr.inria.sed.logo.Logo
        - animator: (optionnal) the .aird file that has a diagram for your model
        - Main method: select xxx.LogoProgramAspect.run(xxx)
        - Main model element path: the LogoProgramImpl
    - *Debug*

The console named "Simple Logo interpreter" will contain your output if you used the GEMOC MessagingSystem, otherwise, `printl` will go to the standard output which is shown by the *Default MessagingSystem console*.

NOTE	you may have to switch between the console in order to retrieve the one with your message.

Define Runtime Data structure

- *new Ecore Modeling Project*
    - project name: *fr.inria.sed.logo.vm.model*
    - Main package name: vm
    - NsUris: `http://www.inria.fr/sed/logo/vm`

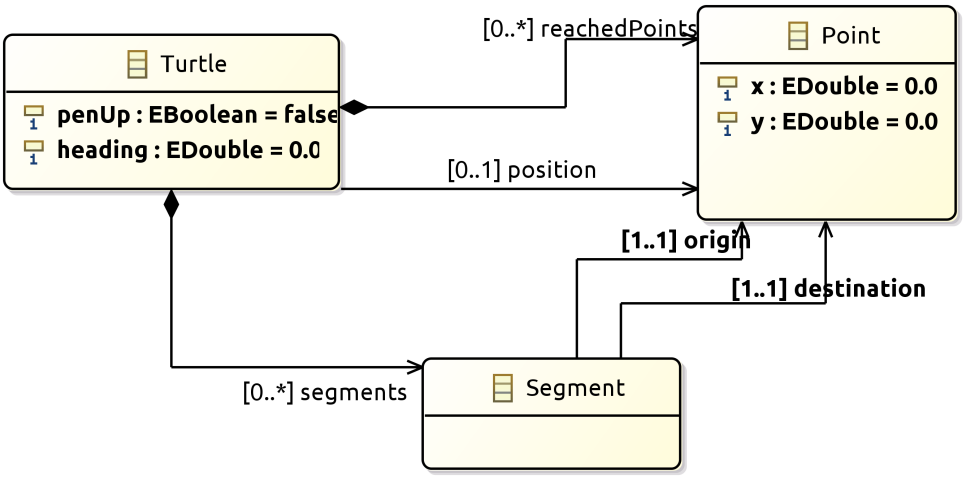
TIP	Installing OCLinEcore allows to write the ecore model in text instead of using the three editor or the graphica editor. In our case, this will help to to copy/paste actions.
	- *Help → Install new software...*
	  - Work with: *Eclipse Repository - http://download.eclipse.org/releases/photon*
	  - get: *OCL Examples and Editors SDK*
	  - proceed to the installation and accept to restart eclipse
	A new editor is now available with a right click on `ecore` files: *Open with → OCLInEcore Editor*.

Create a data structure to capture the runtime state of the turtle running the logo program.

The runtime will be turtle that also store the path it had drawn.

The path is stored as an ordered list of segments.

Some attributes need to be encoded as Double in order to get a simple but realistic simulation.



TIP

Instead of manually creating the various elements in the tree or Sirius editor you can directly use this source and copy/paste using oclinecore editor.

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;

package logo_vm : logo_vm = 'http://fr.inria.sed/logo/logo_vm'
{
    class InterpreterRuntimeContext
    {
        property turtle : Turtle[1] {composes};
    }
    class Turtle
    {
        property reachedPoints : Point[*|1] { ordered composes };
        property position : Point[?];
        property segments : Segment[*|1] { ordered composes };
        attribute penUp : Boolean[1];
        attribute heading : ecore::EDouble[1];
    }
    class Point
    {
        attribute x : ecore::EDouble[1];
        attribute y : ecore::EDouble[1];
    }
    class Segment
    {
        property origin : Point[1];
        property destination : Point[1];
    }
}
```

- right click on the vm.genmodel file → reload...
- rigth click on the root element
- generate Model code

on the plugin.xml of the k3dsa project, add a dependency to *fr.inria.sed.logo.vm.model*.

Link the RuntimeData to the Logo program

Create an "anchor" element in the Logo program Logo.ecore. Ie add an class RuntimeContext and a composition to it from the root model element. This runtimecontext is annotated with "aspect" annotation in order to indicate that it can change during the execution.

NOTE	This is not mandatory for all execution scenarios but will help obtain all GEMOC features

TIP	For some language you may directly weave runtime data in the language ecore. This might be useful to help navigation in the models and data.

in Logo.ecore

```
    class LogoProgram
    {
        property instructions : Instruction[*|1] { ordered composes };
        property runtimeContext : RuntimeContext[?] { composes }
        {
            annotation aspect;
        }
    }

    abstract class RuntimeContext;
```

add a plugin dependencies from *fr.inria.sed.logo.vm.model* to *fr.inria.sed.logo.model*

in VM.ecore

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore#/' ;
import logo : '../../fr.inria.sed.logo.model/model/Logo.ecore#/' ;

package vm : vm = 'http://www.inria.fr/sed/logo/vm'
{
    class InterpreterRuntimeContext extends logo::RuntimeContext
    {
        property turtle : Turtle[1] { composes };
    }
```

regenerate model code of Logo and its VM (IE. from logo.genmodel and vm.genmodel files.)

WARNING	when generating model from vm.genmodel, make sure to correctly reference and reuse the logo.genmodel. Otherwise you'll get 2 copies of the java code for the logo.ecore model that may conflict with each other.

Initialize RuntimeContext on start

In the k3dsa project.

in logoAspects.xtend

```
@Aspect(className=LogoProgram)
class LogoProgramAspect {

    @Step
    @InitializeModel
    def void initializeModel(EList<String> args){
        val context = VmFactory.eINSTANCE.createInterpreterRuntimeContext
        context.turtle = VmFactory.eINSTANCE.createTurtle
        val point = VmFactory.eINSTANCE.createPoint
        point.x = 0
        point.y = 0
        context.turtle.reachedPoints.add(point)
        context.turtle.position = point
        _self.runtimeContext = context
    }
```

Write a simple navigation

TIP

for better performances and cleaner code, the logger accessor can be moved to the context as a "singleton"

```
package fr.inria.sed.logo.k3dsa.logo.vm.aspects

import fr.inria.diverse.k3.al.annotationprocessor.Aspect

import fr.inria.sed.logo.vm.model.vm.InterpreterRuntimeContext
import
org.eclipse.gemoc.commons.eclipse.messagingsystem.api.MessagingSystemManag
er
import
org.eclipse.gemoc.commons.eclipse.messagingsystem.api.MessagingSystem

@Aspect(className=InterpreterRuntimeContext)
class InterpreterRuntimeContextAspect {
    var MessagingSystem internalLogger
    def MessagingSystem logger(){
        if (_self.internalLogger === null) {
            val MessagingSystemManager msManager = new
MessagingSystemManager
            _self.internalLogger =
msManager.createBestPlatformMessagingSystem("Logo","Simple Logo
interpreter")

        }
        return _self.internalLogger
    }
}
```

in logoAspect.xtend

```
@Aspect(className=LogoProgram)
class LogoProgramAspect {
    @Main
    def void run(){
        val context = _self.runtimeContext as InterpreterRuntimeContext
        context.logger.debug("Running "+_self.eResource.URI, "Logo")

        _self.instructions.forEach[i | i.run(_self.runtimeContext as
InterpreterRuntimeContext)]
    }
}

@Aspect(className=Instruction)
class InstructionAspect {
    @Step
    def void run(InterpreterRuntimeContext context){
        context.logger.error("run of " +_self +" should never occur, please write
```

```
method run for this class",
            "Logo")
    }
}

@Aspect(className=Expression)
class ExpressionAspect {
    def Integer eval(InterpreterRuntimeContext context){
        context.logger.error("eval of " +_self +" should never occur, please write
method run for this class",
            "Logo")
        return 0;
    }
}

@Aspect(className=If)
class IfAspect extends ControlStructureInstructionAspect {
    @Step
    def void run(InterpreterRuntimeContext context){
        context.logger.debug("run of " +_self, "Logo")
        if(_self.condition.eval(context) == 1) {
            _self.thenPart.run(context)
        } else {
            _self.elsePart.run(context)
        }
    }
}

@Aspect(className=Constant)
class ConstantAspect extends ExpressionAspect {
    def Integer eval(InterpreterRuntimeContext context){
        context.logger.debug("eval of " +_self, "Logo")
        return _self.integerValue
    }
}
```

NOTE	We put **@Step** only on `run` methods, since we do want the model debugger to allows to stop there. But do not add this annotation on the `eval` methods.

implements eval methods of classes that inherit from Expression

This is quite simple, most of them maps to very simple code in java/xtend.

```
@Aspect(className=Plus)
class PlusAspect extends ExpressionAspect {
    def Integer eval(InterpreterRuntimeContext context){
        return _self.lhs.eval(context) + _self.rhs.eval(context)
    }
}

@Aspect(className=Minus)
class MinusAspect extends ExpressionAspect {
    def Integer eval(InterpreterRuntimeContext context){
        return _self.lhs.eval(context) - _self.rhs.eval(context)
    }
}
```

For boolean expressions, we simpliflied the problem in the metamodel by returning only integer, where 0 is false and 1 is true.

```
@Aspect(className=Equals)
class EqualsAspect extends ExpressionAspect {

    def Integer eval(InterpreterRuntimeContext context){
        if( _self.lhs.eval(context) ==  _self.rhs.eval(context)) return 1
        else return 0
    }
}

@Aspect(className=Greater)
class GreaterAspect extends ExpressionAspect {
    def Integer eval(InterpreterRuntimeContext context){
        if( _self.lhs.eval(context) >  _self.rhs.eval(context)) return 1
        else return 0
    }
}
```

Make the turtle move

Ie. modify the runtime context (turtle, segment, ...)

First add some helpers as aspect directly on the vm.

in fr.inria.sed.logo.k3dsa.logo.vm.aspects.TurtleAspect.xtend

```
package fr.inria.sed.logo.k3dsa.logo.vm.aspects

import fr.inria.diverse.k3.al.annotationprocessor.Aspect

import fr.inria.sed.logo.vm.model.vm.Turtle
import fr.inria.sed.logo.vm.model.vm.VmFactory

@Aspect(className=Turtle)
class TurtleAspect {

    def void rotate(Integer angle) {
        _self.heading = (_self.heading + angle) % 360
    }

    def void move(double dx, double dy){
        // create new Point for destination
        val point = VmFactory.eINSTANCE.createPoint
        point.x = _self.position.x + dx
        point.y = _self.position.y + dy
        _self.reachedPoints.add(point)

        if(!_self.penUp){
            val drawnSegment = VmFactory.eINSTANCE.createSegment
            drawnSegment.origin = _self.position
            drawnSegment.destination = point
            _self.segments.add(drawnSegment)
        }
        _self.position = point
    }

    def void forward(Integer steps){
        val headingAsRadian = Math.toRadians(_self.heading)
        _self.move(_self.scale(steps, Math.sin(headingAsRadian)), _self.scale(steps,
Math.cos(headingAsRadian)))
    }

    /**
     * scale the "steps" expressed using integer by a factor
     */
    def double scale(Integer steps, Double factor){
        return (steps.doubleValue * factor) as Double
    }
}
```

Then use them.

in logoAspects.xtend

```
import static extension fr.inria.sed.logo.k3dsa.logo.vm.aspects.TurtleAspect.*

@Aspect(className=Forward)
class ForwardAspect extends PrimitiveInstructionAspect {
    @Step
    def void run(InterpreterRuntimeContext context){
        context.turtle.forward(_self.steps.eval(context))
    }
}
@Aspect(className=Forward)
class BackwardAspect extends PrimitiveInstructionAspect {
    @Step
    def void run(InterpreterRuntimeContext context){
        context.turtle.forward(- _self.steps.eval(context))
    }
}
@Aspect(className=Left)
class LeftAspect extends PrimitiveInstructionAspect {
    @Step
    def void run(InterpreterRuntimeContext context){
        context.turtle.rotate(- _self.angle.eval(context))
    }
}

@Aspect(className=Right)
class RightAspect extends PrimitiveInstructionAspect {
    @Step
    def void run(InterpreterRuntimeContext context){
        context.turtle.rotate(_self.angle.eval(context))
    }
}
```

Get dedicated custom GUI (using EngineAddon)

NOTE	documentation about engine addon creation https://download.eclipse.org/gemoc/docs/nightly/_contributing.html#_developing_new_engines

There are many ways to create a GUI for the simulator. One of them is to create a language specific engine addon. It will be started automatically when the engine starts. It will then be notified by the engine about any relevant event. It has access to many informations including a full access to the model and runtime data model.

- open the plugin.xml file of the project `fr.inria.sed.logo.xdsml`
  - Right click on the XDSML_Definition (fr.inria.sed.logo.Logo) → New → EngineAddon_Definition
  - Click on the link (blue) _engineAddon_class to create the missing class

        - Package: fr.inria.sed.logo.xdsml.ui.turtleboard
        - Name: TurtleBoardEngineAddon

Due to: https://github.com/eclipse/gemoc-studio-modeldebugging/issues/44 remove import, and then apply quick fix to retrieve the correct import (org.eclipse.gemoc.xdsmlframework.api.engine_addon.IEngineAddon).

- in the TurtleBoardEngineAddon java class
    - Right click in the editor
        - *source → override/implements methods*
        - select `engineStarted`, `engineAboutToDispose`, and `stepExecuted`
        - implement the methods to call a GUI reading the model in the engine
            - copy the simple AWT UI implementation from https://github.com/dvojtise/mde-crashcourse-logo/tree/master/part3-mmfirst-solution/fr.inria.sed.logo.xdsml/src/fr/inria/sed/logo/xdsml/ui/turtleboard also copy the `engineStarted`, `engineAboutToDispose`, and `stepExecuted` content.
            - You can observe in TurtleBoardEngineAddon.java How to access the model and runtime data.

NOTE	Callbacks to addons methods create pauses in the execution. You must take care to not crash in an addon, otherwise the execution will crash too. You must take care to long running process and consider using threads/jobs for them (unless this is an intended behavior of you UI).

In the **modeling workbench**, launch an execution on a simple logo model to obser this simple GUI.

More complexe GUI can be written, for example by creating a view integrated in eclipse.

implement ProcedureCall

add a stack of parameter maps in the runtime context

in the vm.ecore

```
    class InterpreterRuntimeContext extends logo::RuntimeContext
    {
        property turtle : Turtle[1] { composes };
        attribute stack : ParamMap(String, ecore::EIntegerObject)[*|1] { ordered
!unique };
    }
    datatype ParamMap(K, V) : 'java.util.HashMap' { serializable };
```

TIP	You can write this kind of code with generics directly in the tree editor, for this you must open the vm.ecore files with the "sample reflective editor" and in the top menu, then click on *sample reflective editor* and *Show generics*

Add some helpers methods to manipulate this stack.

in InterpreterRuntimeContextAspect.xtend

```
    /* paramMap helpers */
    def void pushParamMap(HashMap<String, Integer> paramMap) {
        _self.stack.add(paramMap)
    }
    def HashMap<String, Integer> peekParamMap(){
        _self.stack.last
    }
    def  HashMap<String, Integer> popParamMap(){
        _self.stack.last
        _self.stack.remove(_self.stack.size -1)
    }
```

Use the parameter map to implement the Procedure Call

in logoAspects.xtend

```
import static extension
fr.inria.sed.logo.k3dsa.logo.vm.aspects.InterpreterRuntimeContextAspect.*

@Aspect(className=ProcCall)
class ProcCallAspect extends PrimitiveInstructionAspect {
    @Step
    def void run(InterpreterRuntimeContext context){
        context.logger.debug("run of " +_self, "Logo")
        val HashMap<String, Integer> params = newHashMap;
        (0..(_self.actualArgs.size-1)).forEach[i |
            val currentArg = _self.actualArgs.get(i).eval(context)
            params.put(_self.declaration.args.get(i).name,currentArg)
        ]
        context.pushParamMap(params)
        _self.declaration.instructions.forEach[instruction | instruction.run(context)]
        context.popParamMap()
    }
}

@Aspect(className=ParameterCall, with=#[InstructionAspect] )
class ParameterCallAspect extends ExpressionAspect {
    def Integer eval(InterpreterRuntimeContext context){
        context.logger.debug("eval of " +_self, "Logo")
        return context.peekParamMap.get(_self.parameter.name);
    }
}
```

Add Sirius Debug support.

This will create a dedicated layer that take into account debug interactions

- Right click on the *fr.inria.sed.logo.xdsml* project → *GEMOC language* → *Create animator project for language*
    - *Add a debug layer to an existing diagram description* → *Next* → *Finish*

TODO main pur java

TODO main GEMOC

mettre au point la semantique

ajout du @Step

ajout d'un context / runtime data

trick par ajout d'un attribut Context à LogoProgram

```
LogoProgram :
    {LogoProgram}
    instructions+=Instruction (  instructions+=Instruction)*
    (runtimecontext = RuntimeContext)?
    ;

RuntimeContext returns RuntimeContext:
    {RuntimeContext}
    'RuntimeContext'
;
```

discussion à propos du model first pou masquer cet aspect de la syntaxe

ajout d’un projet ecore modeling "fr.inria.sed.logo.vm.model"

ajout des concepts

heritage de InterpreterContext vers RuntimeContext

puis convertir en 2 languages avec melange : extended pour activer la timeline ou adapter le MM