



Grant ANR-12-INSE-0011

GEMOC
ANR Project, Program INS

**D4.3.1 - Animation engine Eclipse-based plugins, v0,
OBEO (software)
Task 4.3.1**

Version 0.1

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

Document Control

| | - : | A : | B : | C : | D : |
|--------------------------|------------------------|-----|-----|-----|-----|
| Written by Signature | Yvan Lussaud - Obeo | | | | |
| Approved by Signature | | | | | |

| Revision index | Modifications |
|----------------|---------------|
| A | |
| B | |
| C | |
| D | |

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

Authors

| Author | Partner | Role |
|-----------------|---------|-------------|
| Yvan LUSSAUD | Obeo | Lead author |
| François TANGUY | INRIA | Contributor |
| Jérôme LE NOIR | Thales | Contributor |
| | | |

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

Table of Content

| | |
|--|----|
| 1. Introduction..... | |
| 1.1 Purpose..... | |
| 1.2 Perimeter..... | |
| 1.3 Definitions, Acronyms and Abbreviations..... | |
| 1.4 Summary..... | |
| 2. General design..... | |
| 2.1 EMF debug model..... | |
| 2.2 Communication..... | |
| 2.2.1 Debugger state communication..... | 6 |
| 2.2.2 Data communication..... | 8 |
| 3. Execution engine implementation and integration..... | |
| 3.1 Execution engine sample implementation..... | |
| 3.2 Default debugger/animation implementation..... | |
| 3.3 Specific language debugger/animation implementation..... | |
| 4. EMF integration..... | |
| 4.1 Launch configuration..... | |
| 4.1.1 Launch configuration delegate..... | 10 |
| 4.1.2 Launch configuration tab group..... | 11 |
| 4.1.3 Short cuts..... | 12 |
| 4.2 Model presentation..... | |
| 4.3 Breakpoints..... | |
| 4.3.1 Toggling breakpoints..... | 13 |
| 4.3.2 Breakpoint feedback..... | 13 |
| 5. Sirius Integration..... | |
| 5.1 Launch configuration..... | |
| 5.1.1 Launch delegate..... | 14 |
| 5.1.2 Launch configuration tab group..... | 15 |
| 5.1.3 Launch from Sirius..... | 16 |
| 5.2 Model presentation..... | |
| 5.2.1 Eclipse..... | 16 |
| 5.2.2 Sirius..... | 16 |
| 5.3 Breakpoints..... | |
| 5.3.1 Toggling breakpoints..... | 17 |
| 5.3.2 Breakpoint feedback..... | 17 |
| 6. TFSM example..... | |
| 6.1 Debug configuration..... | |
| 6.2 Runtime data configuration..... | |

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

1. Introduction

1.1 Purpose

This documentation aims to describe Eclipse plugins composing the animation engine component of the GEMOC Studio.

1.2 Perimeter

Source code of the animation engine can be found in the Gemoc git repository:

`git+ssh://scm.gforge.inria.fr/gitroot/gemoc-dev/gemoc-dev.git`

in the following folder:

`fr/obeo/dsl/sirius/animation`

1.3 Definitions, Acronyms and Abbreviations

API: Application Programming Interface.

Eclipse Plugin: an Eclipse plugin is a Java project with associated metadata that can be bundled and deployed as a contribution to an Eclipse-based IDE.

EMF: Eclipse Modeling Framework.

Language Workbench: a language workbench offers the facilities for designing and implementing modeling languages.

Language Designer: a language designer is the user of the language workbench.

Modeling Workbench: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.

1.4 Summary

The Document will cover the general design and software aspects of the animation engine. The document will provide information about integration of the animation engine with the Gemoc execution engine, the EMF tree editors and finally with Sirius modeler.

2. General design

The global architecture of Gemoc is shown in Figure 1. The debugger and animator are configured and the Concrete Syntax Configurator. A generic part is connected with the MSE Engine for MSE capturing and runtime data updates. Finally those information are presented in the Modeling workbench using the Model Animator.

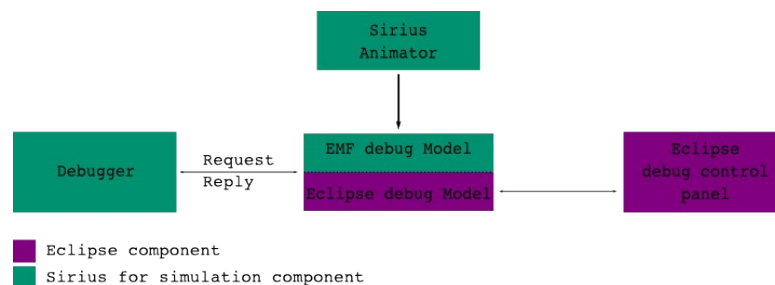


Figure 1 : General design

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

The debugger is a layer on top of the Eclipse debugging framework for EMF integration. It will use EObject as instruction representation. It will also include an other component on top of it dedicated to the animation of Sirius models according to the debugger state and runtime data. The Figure 2 shows the general architecture of the GEMOC Studio.

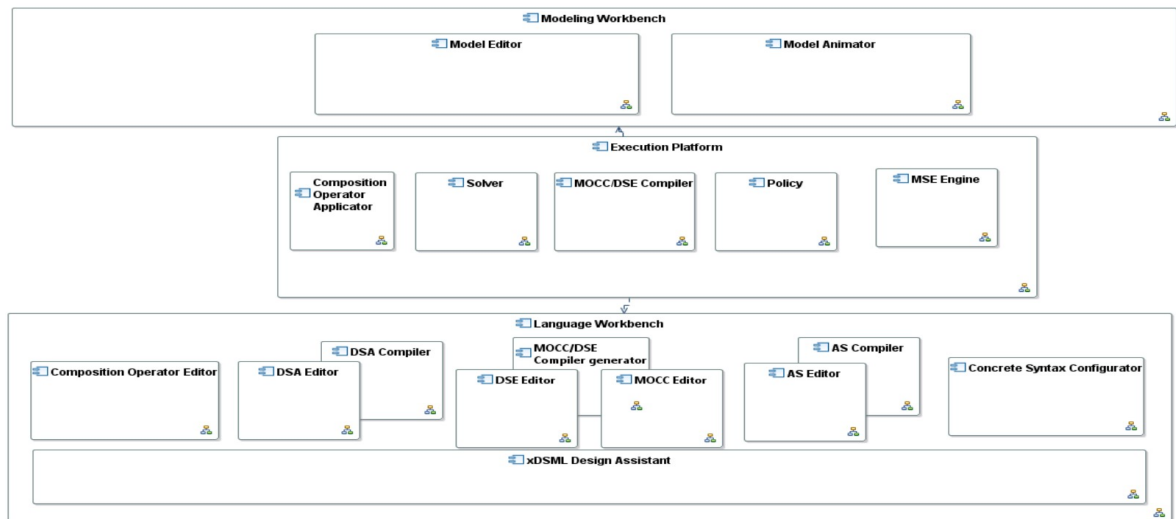


Figure 2 : Architecture of the GEMOC Studio

The central component is a model representing the debugger state. One part belongs to the Eclipse debug framework and is used by Eclipse debug control panel to drive the debugger execution and also display data about the debugger state. The second part is an EMF layer for modeling integration. The communication between the Eclipse debug model and the Eclipse debug control panel is part of the Eclipse debug framework. It is asynchronous but will not be discussed here.

On top of this Eclipse debug model we added an EMF representation of the Eclipse debug model. This EMF model will be used by the Sirius animator component. It could also be used by other EMF tools.

The communication between the debugger and the model is done asynchronously using events. The debugger controls the execution of the underlying execution engine and peeks meaningful data about its internal state.

Sirius Animator is used to present runtime data while the model is being executed. It comes on top of the model presentation when the presentation is done using Sirius.

2.1 EMF debug model

The model is an EMF representation of the Eclipse debug model. It also includes information about the state of the debugger. States are checked by utility methods allowing the modification of the model.

2.2 Communication

This section presents the communication between the debugger and the Eclipse debugging framework.

2.2.1 Debugger state communication

The debug target and its threads implements the following state automata:

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

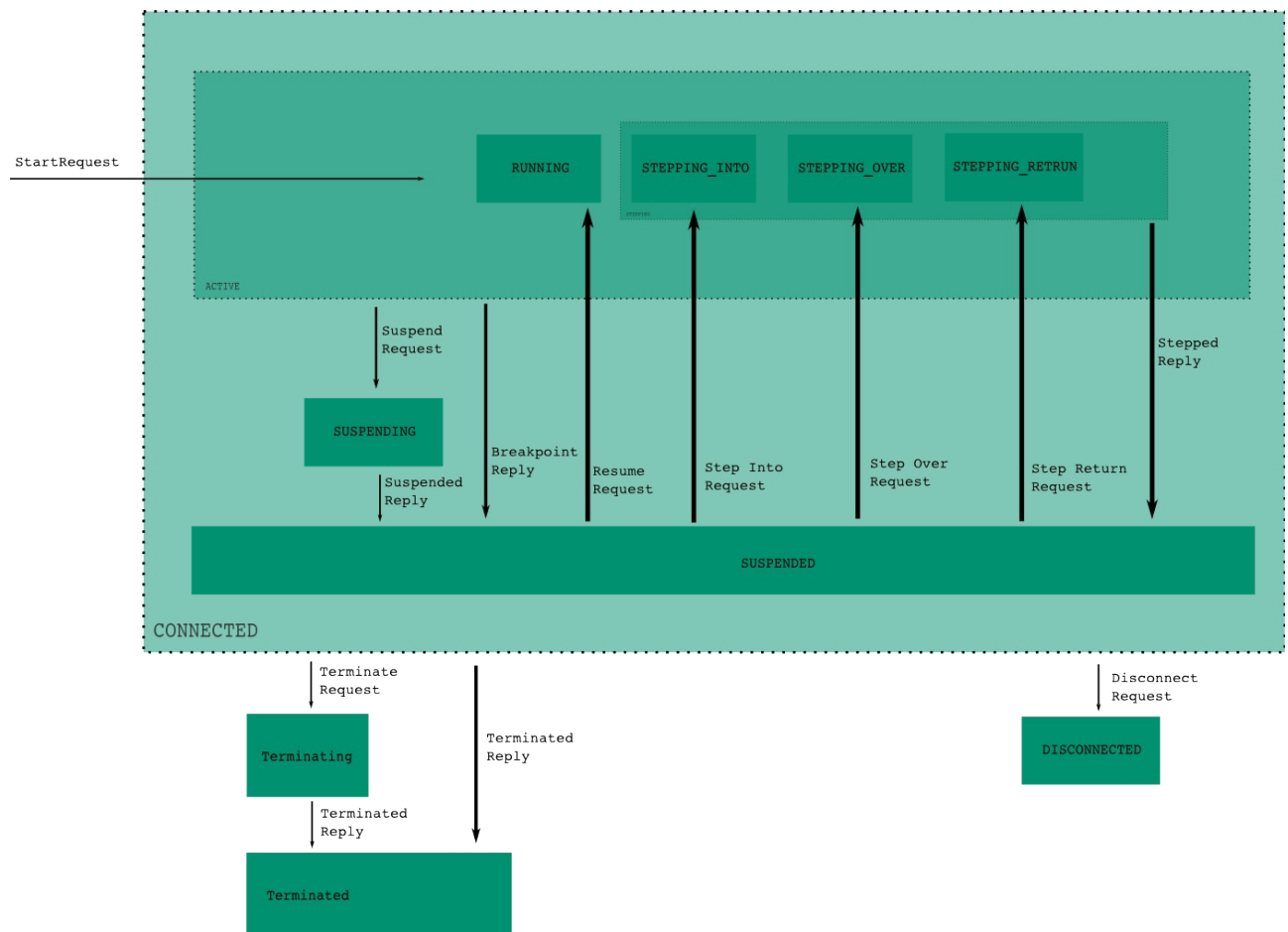


Figure 3 : debugger state automata

The communication is event based. Events from the Eclipse debug framework to the debugger are called “Request” and event from the debugger to the Eclipse debug framework are called “Reply”. As described in the previous section the communication is asynchronous and handled by DSLDebugEventDispatcher.

Requests:

- DisconnectRequest
- StartRequest
- ResumeRequest
- SuspendRequest
- StepIntoRequest
- StepOverRequest
- StepReturnRequest
- TerminateRequest

Replies:

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

- BreakPointReply
- ResumingReply (ResumeRequest acknowledgement after a thread wake up)
- StepIntoResumingReply (StepIntoRequest acknowledgement after a thread wake up)
- StepOverResumingReply (StepOverRequest acknowledgement after a thread wake up)
- StepReturnResumingReply (StepReturnRequest acknowledgement after a thread wake up)
- SteppedReply
- SuspendedReply
- TerminatedReply

2.2.2 Data communication

Data communication is made by the debugger to provide information about the process being debugged. Basically threads existing in the debugged process and stack frame structure. It also includes more specific information like variable attached to each stack frame. This part of the communication is driven by the debugger and is specific to the debugged process /language.

Replies:

- DeleteVariableReply
- PopStackFrameReply
- PushStackFrameReply
- SetCurrentInstructionReply
- SpawnRunningThreadReply
- VariableReply

3. Execution engine implementation and integration

This section will deal with the execution engine implementation and integration with a specific language debugger implementation. It will introduce the default debugger implementation and what parts of the debugger should be implemented by a specific language debugger developer. The implementation and integration has been done for the Gemoc execution engine. For more details on the Gemoc execution engine see D4.2.1.

3.1 Execution engine sample implementation

When running in debug mode the execution engine should be ran in an other thread. A sample implementation is as follow:

```
public void run() {
    EObject currentInstruction = someInit(...);
    if (getDebugger() != null) {
        getDebugger().spawnRunningThread(Thread.currentThread().getName(),
        currentInstruction);
    }
    while (!terminated) {
        if (getDebugger() != null) {
```


| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

```

        terminated = getDebugger().control(Thread.currentThread().getName(),
currentInstruction);
    }
    if (!terminated) {
        currentInstruction = evaluate(currentInstruction);
        terminated = currentInstruction == null;
    }
}
if (getDebugger() != null && !
getDebugger().isTerminated(Thread.currentThread().getName())) {
    getDebugger().terminated(Thread.currentThread().getName());
}
}

```

The debugger is first asked to spawn a running thread. Then, in the main loop, it controls the execution of the thread according to command passed via the Eclipse debug control panel. It may suspend or resume the thread execution. Finally the debugger is asked to terminate the thread spawned at the beginning.

Since the `evaluate()` method might be split across many classes, the call to the `control()` method can be located at the beginning of each `evaluate()` method.

3.2 Default debugger/animation implementation

The default `AbstractDSLDebugger` take care of the communication with the Eclipse debug framework. It also implements a default behavior:

- suspend/resume/terminate on the debug target apply the command to all threads
- terminate on the last thread terminate the debug target

The default implementation also deals with the data and feature communication. But it should be driven by the debugger implementation for a specific language in order to provide specific runtime data. See the next section for more details.

3.3 Specific language debugger/animation implementation

Some parts of the debugger are related to the language being debugged. For instance the threads, stack frames, and variables. Feature support should also be defined by the specific debugger implementation.

The thread can be manipulated by using:

- `IDSLDebugger.spawnRunningThread(String threadName, EObject context)`
- `IDSLDebugger.terminate(String threadName)`

The stack frame and variable manipulation must be done by implementing `IDSLDebugger.updateData(String threadName, EObject instruction)`. The stack frame can be manipulated by using:

- `IDSLDebugger.pushStackFrame(String threadName, EObject context, EObject instruction)`
- `IDSLDebugger.popStackFrame(String threadName)`
- `IDSLDebugger.setCurrentInstruction(String threadName, EObject instruction)`

The variable can be manipulated by using:

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

- IDSLDebugger.variable(String threadName, String declarationTypeName, String name, Object value)
- IDSLDebugger.deleteVariable(String threadName, String name)

4. EMF integration

This section describes the integration needed to be able to follow the execution process in the Eclipse IDE. This integration is based on the default EMF tree editor. It could be adapted to fit your own editor.

4.1 Launch configuration

The launch configuration is used to execute a program from the Eclipse IDE. Detailed documentation on launch configuration can be found in [this article](#). This section will only cover specific aspects for our case.

4.1.1 Launch configuration delegate

In the plugin.xml file declares a new extension in order to activate the launch action set (run/debug action in the tool bar) in all perspective:

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension targetID="*">
    <actionSet id="org.eclipse.debug.ui.launchActionSet">
    </actionSet>
  </perspectiveExtension>
</extension>
```

Next we want our own launch configuration type to debug our DSL:

```
<extension point="org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
    delegate="com.example.dsl.ui.launch.Delegate"
    id="com.example.dsl.ui.launchConfiguration"
    modes="debug"
    name="My DSL launch configuration"
    sourceLocatorId="fr.obeo.dsl.debug.ide.sourceLocator">
  </launchConfigurationType>
</extension>
```

Note that you can add modes for instance “run”.

Now add an image to our launch configuration:

```
<extension point="org.eclipse.debug.ui.launchConfigurationTypeImages">
  <launchConfigurationTypeImage
    configTypeID="com.example.dsl.ui.launchConfiguration"
    icon="icons/launch.png"
    id="com.example.dsl.ui.launchConfigurationImage">
  </launchConfigurationTypeImage>
</extension>
```

At this point we need to have a look at the implementation of `com.example.dsl.ui.launch.Delegate`. This class extends the default implementation provided by the DSL debugger framework: `fr.obeo.dsl.debug.ide.ui.launch.AbstractDSLLaunchConfigurationDelegateUI`. You will have to implement the following methods:

```
@Override
protected IDSLDebugger getDebugger(ILaunchConfiguration configuration,
    DSLDebugEventDispatcher dispatcher, EObject firstInstruction,
    IProgressMonitor monitor) {
```

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

```

MyEngine engine = new MyEngine();
engine.init(firstInstruction, ...);
MyDebugger res = new MyDebugger(dispatcher, engine);
engine.setDebugger(res);
return res;

```

}

This method instantiates the execution engine and the debugger and connect them to the DSLDebugEventDispatcher.

When creating a new launch configuration we need to get the first instruction either form an ISelection or an IEditorPart:

```

@Override
protected EObject getFirstInstruction(ISelection selection) {
    final IResource resource = getLaunchableResource(selection);
    return ...;
}

@Override
protected EObject getFirstInstruction(IEditorPart editor) {
    final EObject res;

    ISelection selection = PlatformUI.getWorkbench().getActiveWorkbenchWindow().
        getSelectionService().getSelection();
    if (selection instanceof IStructuredSelection
        && ((IStructuredSelection) selection).getFirstElement() instanceof EObject
        && is...(((EObject) ((IStructuredSelection) selection)
            .getFirstElement())) {
        res = (EObject) ((IStructuredSelection) selection)
            .getFirstElement();
    } else {
        final IResource resource = getLaunchableResource(editor);
        res = ...;
    }

    return res;
}

```

This is only a sample code. If null is returned then the launch configuration tab group will be shown in order to prompt the user for the first instruction. For more details about the tab group see the next section.

4.1.2 Launch configuration tab group

The launch configuration tab group configures tabs shown in the launch configuration dialog. This part is highly related to the initialization of the debug engine, but the DSL debugger framework provide a default implementation as a starting point.

First declare a new extension:

```

<extension point="org.eclipse.debug.ui.launchConfigurationTabGroups">
    <launchConfigurationTabGroup
        class="com.example.dsl.ui.launch.TabGroup"
        id="com.example.dsl.ui.launchConfigurationTabGroup"
        type="com.example.dsl.ui.launchConfiguration">
        <launchMode

```

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

```

        mode="debug"
        perspective="org.eclipse.debug.ui.DebugPerspective">
    </launchMode>
</launchConfigurationTabGroup>
</extension>

```

A default implementation of `com.example.dsl.ui.launch.TabGroup` extending `AbstractLaunchConfigurationTabGroup`:

```

@Override
public void createTabs(ILaunchConfigurationDialog dialog, String mode) {
    setTabs(new ILaunchConfigurationTab[] { new DSLLaunchConfigurationTab(new String[]
        {"ext",}), new CommonTab(), });
}

```

“ext” should be replaced by the extension file corresponding to your DSL.

4.1.3 Short cuts

This section is optional but ease the use of the debugger launch configuration. It adds shortcuts to the launch configuration.

Add the following extension:

```

<extension point="org.eclipse.debug.ui.launchShortcuts">
    <shortcut
        class="com.example.dsl.ui.launch.Delegate"
        description="My debug"
        icon="icons/launch.png"
        id="com.example.dsl.ui.shortcut"
        label="My debug"
        modes="debug">
        <contextualLaunch>
            <enablement>
                <count value="1">
            </count>
            <iterate>
                <test property="org.eclipse.debug.ui.matchesPattern"
                    value="ext">
            </test>
            </iterate>
            </enablement>
        </contextualLaunch>
        <configurationType id="com.example.dsl.ui.launchConfiguration">
        </configurationType>
    </shortcut>
</extension>

```

“ext” should be replaced with the extension file corresponding to your DSL. And of course the selection strategy can be changed.

4.2 Model presentation

The debug model presentation is responsible for GUI representation of the debug model. It provides icons and labels. It also opens editor when a breakpoint is hit, or an instruction must be displayed for a given stack frame The DSL debugger provides a default implementation for generated EMF editors integration.

It can be used via the following extension:

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

```

<extension point="org.eclipse.debug.ui.debugModelPresentations">
  <debugModelPresentation
    class="fr.obeo.dsl.debug.ide.ui.DSLDebugModelPresentation"
    id="com.example.dsl.ui.debugModel">
  </debugModelPresentation>
</extension>

```

It can also be extended for specific needs.

4.3 Breakpoints

This section will present the breakpoint action and feedback in order to improve the debugging experience.

4.3.1 *Toggling breakpoints*

Then we need to add a command to the popup menu:

```

<extension point="org.eclipse.ui.menus">
  <menuContribution
    allPopups="false"
    locationURI="popup:com.example.dsl.editor.presentation.MyDSLEditorID">
    <command commandId="com.example.dsl.ui.toggleBreakpointCommand"
      icon="icons/breakpoint.gif"
      id="com.example.dsl.ui.toggleBreakpointCommand"
      label="Toggle My DSL breakpoint"
      style="push"
      tooltip="Toggle a My DSL breakpoint">
    </command>
  </menuContribution>
</extension>
<extension point="org.eclipse.ui.handlers">
  <handler
    class="com.example.dsl.ui.command.ToggleBreakpointHandler"
    commandId="com.example.dsl.ui.toggleBreakpointCommand">
  </handler>
</extension>
<extension point="org.eclipse.ui.commands">
  <command id="com.example.dsl.ui.toggleBreakpointCommand"
    name="Toggle My DSL breakpoint">
  </command>
</extension>

```

Now lets have a look at the `com.example.dsl.ui.command.ToggleBreakpointHandler`:

4.3.2 *Breakpoint feedback*

The visual feedback of installed breakpoints is done by decorating `ILabelProvider`:

```

final AdapterFactoryLabelProvider labelProvider = new
AdapterFactoryLabelProvider(adapterFactory);
viewer.setLabelProvider(new DecoratingColumLabelProvider(labelProvider, new
DSLLabelDecorator(editingDomain.getResourceSet(), Delegate.MODEL_ID));

```

The decorator reacts to added and removed breakpoints as well as enable and disable breakpoints. The `ResourceSet` is used to retrieve instruction from the URI in the editor context.

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

5. Sirius Integration

This section covers the integration for Sirius editors. Most aspects are the same as the previous section but will also be presented here. This will allow to use this section with no knowledge of the previous section.

5.1 Launch configuration

The launch configuration is used to execute a program from the Eclipse IDE. Detailed documentation on launch configuration can be found in this article. This section will only cover specific aspects for our case.

5.1.1 Launch delegate

In the plugin.xml file declare a new extension in order to activate the launch action set (run/debug action in the tool bar) in all perspective:

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension targetID="*">
    <actionSet id="org.eclipse.debug.ui.launchActionSet">
    </actionSet>
  </perspectiveExtension>
</extension>
```

Next we want our own launch configuration type to debug our DSL:

```
<extension point="org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
    delegate="com.example.dsl.sirius.ui.launch.MokaDelegate"
    id="com.example.dsl.sirius.ui.launchConfiguration"
    modes="debug"
    name="Moka Sirius launch configuration"
    sourceLocatorId="fr.obeo.dsl.debug.ide.sourceLocator">
  </launchConfigurationType>
</extension>
```

Note that you can add modes for instance “run”. The source locator is the same for all DSL debugger.

Now add an image to our launch configuration:

```
<extension point="org.eclipse.debug.ui.launchConfigurationTypeImages">
  <launchConfigurationTypeImage
    configTypeID="com.example.dsl.sirius.ui.launchConfiguration"
    icon="icons/launch.png"
    id="com.example.dsl.sirius.ui.launchConfigurationImage">
  </launchConfigurationTypeImage>
</extension>
```

At this point we need to have a look at the implementation of `com.example.dsl.sirius.ui.launch.Delegate`. This class extends the default implementation provided by the DSL debugger framework: `fr.obeo.dsl.debug.ide.sirius.ui.launch.AbstractDSLLaunchConfigurationDelegateUI`. You will have to implement the following methods:

```
@Override
protected IDSLDebugger getDebugger(ILaunchConfiguration configuration,
    DSLDebugEventDispatcher dispatcher, EObject firstInstruction,
    IProgressMonitor monitor) {
    MyEngine engine = new MyEngine();
    engine.init(firstInstruction, ...);
    MyDebugger res = new MyDebugger(dispatcher, engine);
    engine.setDebugger(res);
}
```

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

```

        return res;
    }

```

This method instantiates the execution engine and the debugger and connect them to the DSLDebugEventDispatcher.

When creating a new launch configuration we need to get the first instruction either form an ISelection or an IEditorPart:

```

@Override
protected EObject getFirstInstruction(ISelection selection) {
    final IResource resource = getLaunchableResource(selection);
    return ...;
}

@Override
protected EObject getFirstInstruction(IEditorPart editor) {
    final EObject res;

    ISelection selection = PlatformUI.getWorkbench().getActiveWorkbenchWindow().
getSelectionService().getSelection();
    if (selection instanceof IStructuredSelection
        && ((IStructuredSelection) selection).getFirstElement() instanceof EObject
        && is...(((EObject) ((IStructuredSelection) selection)
            .getFirstElement())) {
        res = ((EObject) ((IStructuredSelection) selection)
            .getFirstElement());
    } else {
        final IResource resource = getLaunchableResource(editor);
        res = ...;
    }

    return res;
}

```

This is only a sample code. If null is returned then the launch configuration tab group will be shown in order to prompt the user for the first instruction. For more details about the tab group see the next section.

5.1.2 Launch configuration tab group

The launch configuration tab group configure tabs shown in the launch configuration dialog. This part is highly related to the initialization of the debug engine, but the DSL debugger framework provide a default implementation as a starting point.

First declare a new extension:

```

<extension point="org.eclipse.debug.ui.launchConfigurationTabGroups">
    <launchConfigurationTabGroup
        class="com.example.dsl.sirius.ui.launch.MokaTabGroup"
        id="com.example.dsl.sirius.ui.launchConfigurationTabGroup"
        type="com.example.dsl.sirius.ui.launchConfiguration">
        <launchMode
            mode="debug"
            perspective="org.eclipse.debug.ui.DebugPerspective">
        </launchMode>
    </launchConfigurationTabGroup>

```

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

</extension>

A default implementation of `com.example.dsl.sirius.ui.launch.TabGroup` extending `AbstractLaunchConfigurationTabGroup`:

@Override

```
public void createTabs(ILaunchConfigurationDialog dialog, String mode)
{
    setTabs(new ILaunchConfigurationTab[] { new
DSLLaunchConfigurationTab(new String[]{"ext",}), new CommonTab(), });
}
```

“ext” should be replaced by the extension file corresponding to your DSL.

5.1.3 Launch from Sirius

In order to be able to launch the debugger, our DSL debug framework provides a `IEExternalJavaAction` that creates or reuse a launch configuration and then launch it in debug mode. You should extends the `fr.obeo.dsl.debug.ide.sirius.ui.action.AbstractDebugAsAction` class. The method `getFirstInstruction()` should be implemented in order to retrieve the first instruction from the current selection. If null is returned then the launch configuration tab group will be shown in order to prompt the user for the first instruction.

Your instance of `IEExternalJavaAction` must be declared via the following extension point in order to be used in your view point description:

```
<extension point="org.eclipse.sirius.externalJavaAction">
    <javaActions actionClass="com.example.dsl.sirius.ui.action.MyDSLDebugAs"
        id="com.example.dsl.sirius.ui.action.mokaDebugAs">
    </javaActions>
</extension>
```

You can then use it in a tool or a popup menu action. Check the Sirius documentation for further details.

5.2 Model presentation

5.2.1 Eclipse

The debug model presentation is responsible for GUI representation of the debug model. It provides icons and labels. It also opens editor when a breakpoint is hit, or an instruction must be displayed for a given stack frame. The DSL debugger provides a default implementation for generated EMF editors integration.

It can be used via the following extension:

```
<extension point="org.eclipse.debug.ui.debugModelPresentations">
    <debugModelPresentation
        class="fr.obeo.dsl.debug.ide.sirius.ui.DSLDebugModelPresentation"
        id="com.example.dsl.sirius.ui.debugModel">
    </debugModelPresentation>
</extension>
```

It can also be extended for specific needs.

5.2.2 Sirius

Sirius allows you to visualize models with a wide range of visual features. In order to make it possible for debugger feedback, a Java service class can be extended. Your implementation of `fr.obeo.dsl.debug.ide.sirius.ui.services.AbstractDSLDebuggerServices` can then be imported and used to customize the style of the instruction being debugged. The service method to use is `isCurrentInstruction(EObject instruction)`. Check the [Sirius documentation](#) for further details.

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

5.3 Breakpoints

This section will present the breakpoint action and feedback in order to improve the debugging experience.

5.3.1 *Toggling breakpoints*

The DSL debug framework provides an `IExternalJavaAction` to toggle breakpoints. You should extend `fr.obeo.dsl.debug.ide.sirius.ui.action.AbstractToggleBreakpointAction` and declare the action via the following extension:

```
<extension point="org.eclipse.sirius.externalJavaAction">
  <javaActions
    actionClass="com.example.dsl.sirius.ui.action.MyDSLToggleBreakpointAction"
    id="com.example.dsl.sirius.ui.action.MyDSLToggleBreakpointAction">
  </javaActions>
</extension>
```

You can then use it in a tool or a popup menu action. Check the [Sirius documentation](#) for further details.

5.3.2 *Breakpoint feedback*

As we changed the style of the current instruction we might want to change the style of instructions used as breakpoints. This is done the same way via the same service class. Extend the `fr.obeo.dsl.debug.ide.sirius.ui.services.AbstractDSLDebuggerServices` if not already done and import it in your representation description. You can then use `hasEnabledBreakpoint(EObject instruction)` and `hasDisabledBreakpoint(EObject instruction)` to add a decorator for instance or change the style of the instruction in any way you want. Check the [Sirius documentation](#) for further details.

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

6. TFSM example

The TFSM language, provided in the Gemoc studio, uses Sirius editor. This section will present two aspects of the animation in the editor configuration. The first one is the debug control and feedback as described in previous sections. The second one is the presentation of runtime data.

6.1 Debug configuration

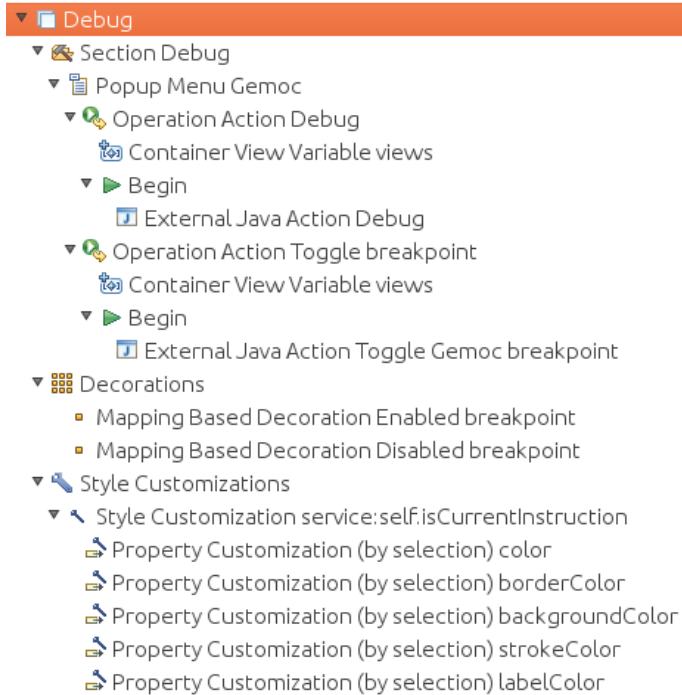


Figure 4 : Debug configuration

The configuration of the Figure 4 renders as shown in the Figure 5.

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |

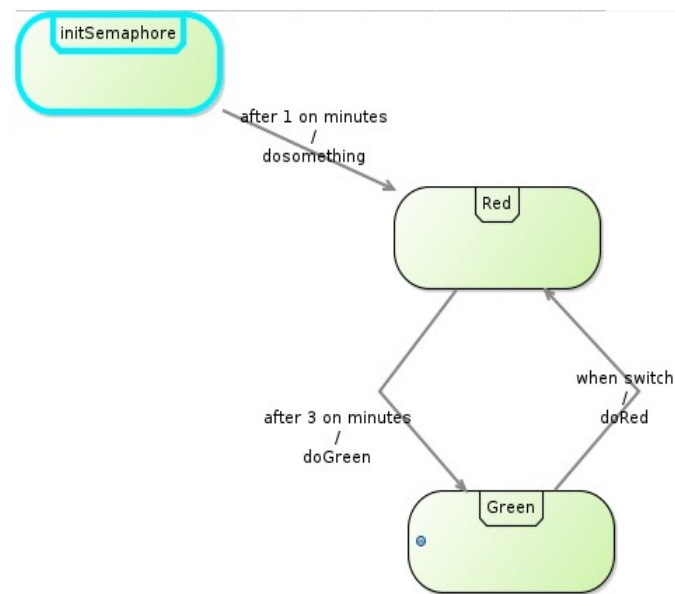


Figure 5 : Debug in the editor

As you can see in Figure 5, the initial state is active and there is a breakpoint on the green state.

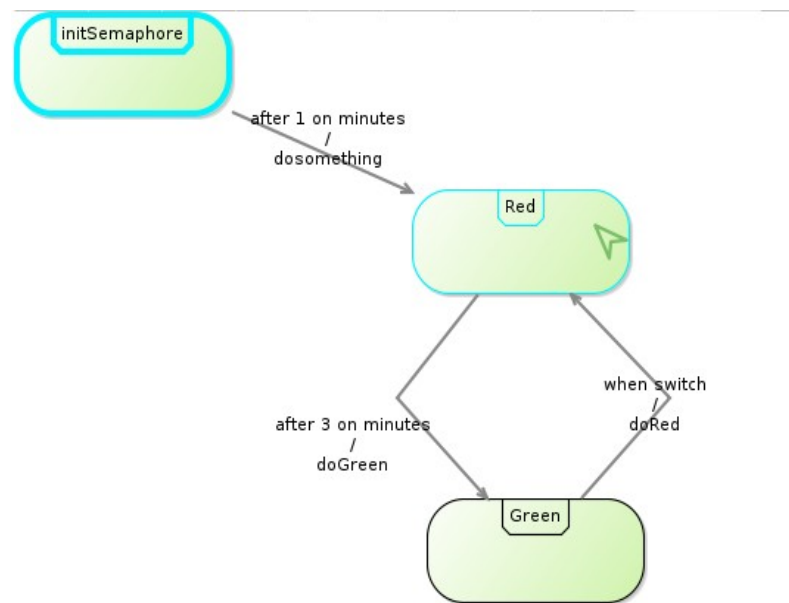
6.2 Runtime data configuration

Runtime data are data manipulated by the engine while executing. It can be interesting to visualize those data. In the TFSM example the current state will be decorated with an arrow as shown on Figure 7.

- ▼ Animator
 - ▼ Decorations
 - ▣ Mapping Based Decoration Current State
 - ▼ Style Customizations
 - ▼ Style Customization service:self.hasBeenActivated
 - ▣ Property Customization (by selection) borderColor

Figure 6 : Runtime data configuration

| | |
|---|------------------|
| ANR INS GEMOC / Task 4.3.1 | Version: 0.1 |
| Animation engine Eclipse-based plugins, v0, OBEO (software) | Date: 12/11/2014 |
| D4.3.1 | |



| | | |
|---|----------------|---------------------|
| Properties <> Interpreter Problems Gemoc Engines Status 1 | | |
| Semaphore | | |
| | Property | Value |
| | TFSM Semaphore | |
| | Current State | State Red |
| | Initial State | State initSemaphore |
| | Name | Semaphore |

Figure 7 : Runtime data in the editor