Tutorial 1.3 : Make Agents Yourself 0.9.11

http://www.irit.fr/PERSONNEL/SMAC/noel/may/

Victor NOËL

January 18, 2010

Contents

1 Introduction

In this tutorial, we are going to see an introduction to MAY

This tool was made to build application-specific models of agent and to generate dedicated framework providing them. Such framework would then be usable to program agents complient with the realised model of agent.

More specifically, a model of agent is realised by an architecture divided in two levels: "container" (the how), hidden to the user of the framework and "application" (the what) expression what the agent does. These two levels links μ-composants together to form a μ-architecture realising the model.

Furthermore, such agents have the possibility to self-adapt at runtime: currently, this means that a μ-composant is able to change the implementation of another μ-composant of the μ-architecture.

In practice, these models are described with the help of the μ-ADL description language, and from this description, the JAVA code needed to implement the μ-composants is generated. The μ-architecture itself is generated and is directly usable by setting the implementation of its μ-composants.

This tool being experimental, several functionalities described are are subject to change in the futur. If possible, this will be stated in this document.

1.1 Requirements

To use MAY, the *Eclipse Classic IDE* is needed. It can be found at `http://www.eclipse.org/`.

1.2 Installation

The installation is done from the *Software Update* in *Eclipse* by adding the following *Update Sites*:

- `http://www.irit.fr/PERSONNEL/SMAC/noel/may/update/`
- `http://oawbranch.pluginbuilder.org/releases/p2-updateSite/`

Choose to install *Make Agents Yourself*.

1.3 Upgrades

In the future, upgrades will be available, just use the *Update* feature of *Eclipse*.

2 Typical Development Process of an Agent with MAY

As explained before, MAY is a tool usable to description μ-architectures based on μ-composants to generate a framework that will be distributed to application developers.

Several phases in the development can be distinguished:

- Description of the μ-composants and μ-architectures:
  - Create a MAY project
  - Describe the model with μ-ADL
  - Generate the corresponding JAVA code
- Implantation of the μ-composants with JAVA
- Implantation of a *Factory* with JAVA
- Generation of the distributable framework in the form of a JAR file

The following sections will describe in detail these development phases. We will see the tools MAY adds to *Eclipse*, a description of μ-ADL and some advice to develop an agent and its μ-composants.

3 Added Tools to *Eclipse*

The tools proposed by MAY are the following:

- Creation of a MAY project
- Creation of a textual μ-ADL description file
- Generation of the corresponding code in JAVA

A graphical editor is currently being worked on.

3.1 Creation of a Project

A MAY project is just a JAVA *Eclipse* project with supplementaty dependencies, an empty μ-ADL desciption file and the required directories already created.

To create a new project, go to *File* menu, *New*, *Project...*, *Make Agents Yourself* category and choose *MAY Project*. Follow the instruction.

3.2 Creation of a Textual μ-ADL Description File

The description of a model of agent is represented by a text file containing the description of μ-composants and μ-architectures with μ-ADL (presented Sect. 4). The extension of these files is `muadlt`. The editor features syntaxic coloration, completion and validation of μ-architectures.

To create a new μ-ADL file, go to *File*, *New*, *Others...*, *Make Agents Yourself* category and choose *MuADL Text file*. Follow the instructions.

3.3 Generation of the corresponding code in JAVA

This action generates the classes required for the implementation of the μ-composants described in the μ-ADL file (the `QuasiComponent`), the class representing the mediator of the μ-architecture and the class facade representing an agent complient with the μ-architecture.

To generate the corresponding code of a μ-ADL file (a file ending in `.muadlt`), do a right click on it and in the submenu *Make Agents Yourself*, choose *Generate QC and Architecture*.

The generated code will be in the `src-gen` directory.

It is important to notice that the content of this directory must never be modified.

4 μ-ADL: Description Language for Agent Architectures

As said before, MAY, using μ-ADL, make possible to describe:

- μ-composants that:
  - Provides operations
  - Requires (from the μ-architecture) operations
  - Requires (from the μ-architecture) the possibility to change other μ-composants
  - Have a state that must be persisted by the μ-architecture when their implementation is changed at runtime

μ-ADL does not permit to specify how they will be implemented (no information of the constructors, the class attributes or private methods)

- μ-architectures containing μ-composants and divided in two levels
- Datatypes use by μ-composants and corresponding to JAVA types (classes or interfaces).

We detail here how to describe these elements.

4.1 DataTypes

To refer to JAVA types, we give them a name in the μ-ADL file.

The syntax is as following:

```
DataType AliasLocal : package.NomClass
DataType AliasLocal : package.NomClass<Generique>
DataType AliasLocal : package.NomClass<package.Generique>

// examples
DataType String : String
DataType Int : int
DataType Msg : my.package.Msg
DataType ListString : java.util.List<String>
DataType MailBox : java.util.List<my.package.Msg>
```

Currently, no verification is done on the existence of these classes

It is important to understand that DataTypes are just aliases, they can't be used inside generics parameters.

4.2 μ-composants

Elements of a μ-composant:

- A name
- Provided operations (`provided`)
- Required operations (`required`)
- Attributes that must be persisted between runtime replacement of implementations (`persistent`)
- A set of μ-composants whose implementation could be changed at runtime (`change`)

This point is likely to change in the future

The μ-composants operations are similar to JAVA methods: they have a name, ordonned typed parameters and a return type. To refer to a method, it is only needed to know its name, its ordonned parameter types and its return type.

UAn example of μ-composant:

```
MuComponent Message {
  package a.test.package
  provided send(mess : String)
  provided receive() : String
  persistent mailbox : MailBox
  required do(String) : Int
  change LifeCycle
}
```

4.3 μ-architectures

A μ-architecture have:

- A name
- A set of μ-composants in the container level (`container`)
- A set of μ-composants in the application level (`application`)
- A set of visibility modifiers on th μ-composants operations (`visibility`)

The μ-composants can be specified as changeable (with `changeable`) by other μ-composants. μ-composants can only change μ-composants of the same level.

By default, a μ-composant operation of the application level is visible by every μ-composants of the μ-architecture, and a μ-composant operation of the container level is only visible by the μ-composants of the container level. This fact can be changed by specifying (with `application` visibility) that an operation of the container level is visible in the application level.

An operation of the μ-architecture can be specified as being available from outside of the agent (with `external`).

This is susceptible to change in the future

An example of μ-architecture:

```
MuArchitecture MyAgent {
  package another.test.package
  container {
    changeable LifeCycle
    Message
  }
  application {
    Behavior
  }
  visibility {
    external Message.send(String)
    application Message.receive() : String
  }
}
```

Notice that :

- If a μ-composant `A` requires an operation `0`, then a μ-composant providing `0` must be present in the μ-architecture.
- Requirements are only validated in the μ-architecture.
- Two μ-composants can't provide the same operation.

4.4 Complementaty Informations

μ-ADL allows for comments :

```
// a comment
/*
a multi-line
comment
*/
```

5 Implantation of μ-composants, use of μ-architectures and Framework Distribution

After the JAVA code from a μ-ADL file was generated (see 3.3), μ-composants implementation and μ-architectures instantiation is possible.

The generation result in a set of JAVA classes situated in the `src-gen` directory. To implement μ-composants, we will write JAVA classes (typically in the `src` directory) that extends some of these generated classes.

5.1 μ-composants Implantations

For each μ-composant `X` defined in the μ-ADL file, MAY generates a class `QuasiComponentX` and a class `AdapterX`. The first must be extended to implement `X`.

For example, there is the two generated classes for a component LifeCyle:

```
public abstract class QuasiComponentLifeCycle
    extends QuasiComponent<AdapterLifeCycle>  {
  abstract public void suicide();
}

public interface AdapterLifeCycle extends Adapter {
  public void cycle(java.lang.String arg0);
  public java.lang.String receive();
}
```

Thus, by extending `QuasiComponentLifeCycle`, one can use the required methods of the μ-composant by using the architecture through the method `this.architecture()`.

There is an example of this μ-composant:

```
public class LifeCycle extends QuasiComponentLifeCycle {
  private boolean alive = true;

  public void start() {
    Runnable me = new Runnable() {
      public void run() {
        while (alive) {
          String msg = architecture().receive();
          architecture().cycle(msg);
        }
      }
    };
    new Thread(me).start();
  }

  @Override
  public void suicide() {
    alive = false;
  }
}
```

The important points to notice are:

- A method `void start()` is present to execute instructions at the start of the agent
- `this.architecture()` gives access to required methods through the μ-architecture

There is an example of a generated μ-composant with a persistent state:

```
public abstract class QuasiComponentMessage
    extends QuasiComponent<AdapterMessage>
    implements WithPersistentState<PersistentStateMessage> {

  abstract public void send(java.lang.String message);
  abstract public java.lang.String receive();
}

public interface AdapterMessage extends Adapter {}
```

When implementing it, one has access to the methods neede for the handling of the persistent state (automatically called by the architecture if the μ-composant implementation is changed at runtime).

There is an example of implementation for this μ-composant:

```
public class Message extends QuasiComponentMessage {

  private ArrayBlockingQueue<String> messageBox =
    new ArrayBlockingQueue<String>(100);

  @Override
  public void send(String message) {
    try {
      messageBox.put(message);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  @Override
  public PersistentStateMessage getPersistentState() {
    return new PersistentStateMessage(
                new ArrayList<String>(messageBox)
          );
  }

  @Override
  public void setPersistentState(PersistentStateMessage state) {
    messageBox =
      new ArrayBlockingQueue<String>(10, false, state.messageBox);
    System.out.println("Message Component started with "+
                        messageBox.size()+ " messages");
  }

  @Override
  public String receive() {
    try {
      return messageBox.take();
    } catch (InterruptedException e) {
```

```
      e.printStackTrace();
      return "";
    }
  }

  @Override
  public void start() {}
}
```

Notice that we use an object of type `PersistentStateMessage` to store and read the persistent state.

There is an example of a generated μ-composant changing another μ-composant implementation:

```
public abstract class QuasiComponentAdaptation
    extends QuasiComponent<AdapterAdaptation>  {
  abstract public void adapt();
}

public interface AdapterAdaptation extends Adapter {
  public void change(QuasiComponentMessage _comp);
}
```

Notice that a `void change(QuasiComponentMessage _comp)` method is present to give access to the replacement functionalities of the μ-architecture.

There is an example of implementation of this μ-composant:

```
public class Adaptation extends QuasiComponentAdaptation {

  @Override
  public void start() {}

  @Override
  public void adapt() {
    architecture().change(new Message());
  }
}
```

We can also see here for the first time the instantiation of a μ-composant.

When calling `change(...)`, the μ-architecture will get the current state of `Message` and will inject it in the new implementation with the methods we defined previously in `Message`.

5.2 μ-architecture Use

To use a μ-architecture is to create an agent complying to this μ-architecture by instantiating each of its μ-composants.

For this, the generated code of a μ-architecture `X` consists into 2 classes: `XMediator` and `X`. The first one is the implementation of the μ-architecture itself, and the second one is a facade providing the external methods of the agent.

This will change in the future.

Thus, to create an agent complient with a μ-architecture `SimpleAgent` with 4 μ-composants `Adaptation`, `Message`, `LifeCycle` and `Behavior` is done like this:

```
SimpleAgent agent1 = new SimpleAgent(
                         new Message(),
                         new LifeCycle(),
                         new Adaptation(),
                         new Behavior());
agent1.start();
```

Notice the call to `start()` that starts each of the μ-composants and thus the agent.

5.3 *Factory*

One of the objectives of MAY being to reduce the development effort by distributing models of agents ready to use, it is necessary to be able to provide partially instantiated μ-architecture where some of the μ-composants implementation are already fixed. These will then be directly usable by users by choosing the remeaning μ-composant implementations.

This is done by using à *Factory*, which currently must be built by hand.

A factory is only a class providing one (or several) static methods instantiating some μ-composants and an agent. For example:

```
public class MySimpleAgent {
  public static SimpleAgent create(QuasiComponentBehavior behavior) {
    SimpleAgent agent = new SimpleAgent(new Message(),
                                        new LifeCycle(),
                                        new Adaptation(),
                                        behavior);
    agent.start();
    return agent;
  }
}
```

Thus, to create an agent, one just need to know `MySimpleAgent.create(...)`.

For example:

```
public class Main {
  public static void main(String[] args) {
    SimpleAgent agent1 = create(new Behavior2("agent1"));
    SimpleAgent agent2 = create(new Behavior("agent2", agent1));

    agent2.send("hello");
  }
}
```

5.4 Framework Distribution

The created classes in the previous sections form a framework usable by a user of the model of agent to create agents.

Delivering them to such an user is to export the *Eclipse* project as a JAR using its export facilities: right-click on the project, choose *Export...*, in the *Java* category, choose *JAR File* and follow the instructions.

The resulting JAR file will only need the `fr.irit.smac.muadl.impl_X.X.X.jar` file available of the MAY website. For example they can both be adde to a classic*Eclipse* JAVA project as dependencies.