Tutoriel 1.2 : Make Agents Yourself 0.9.9

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

Victor NOËL

18 janvier 2010

Table des matières

1 Introduction

Nous allons voir dans ce tutoriel une introduction à l'outil MAY.

Cet outil a pour vocation la construction de modèles d'agents « à la carte » et la génération d'APIs dédiées à ces modèles. De telles API pourront ensuite être redistribuées pour programmer des agents conformes aux modèles.

Plus spécifiquement, un modèle d'agent est une « machine abstraite » ou « conteneur » qui définit comment un agent opère (définit sa « sémantique ») à partir de mécanismes internes. Un modèle d'agent est composé d'un niveau « container » (le comment) caché à l'utilisateur de l'API et d'un niveau « application » qui exprime ce que l'agent fait (le quoi). Ces deux niveaux lient des μ-composants entre eux et forment une μ-architecture à laquelle les agents se conformeront.

De plus, ces agents possèdent la capacité de s'auto-adapter en cours d'exécution : pour le moment, cela se traduit par la possibilité, pour un μ-composant, de changer l'implantation d'un autre μ-composant de la μ-architecture.

En pratique, ces modèles seront décrit à l'aide du langage μ-ADL, puis, à partir de cette description, le code JAVA nécessaire à l'implantation de ces modèles sera généré.

Cet outil étant en phase expérimentale, un certain nombre de comportements et fonctionnalités décrits ici seront susceptibles de changer dans le futur. Si possible, cela sera précisé tout au long de ce document.

1.1 Prérequis

Pour utiliser MAY, il vous suffit d'avoir une installation d'*Eclipse Classic 3.4.2* que l'on peut trouver (au moment de l'écriture du tutoriel) à cette adresse : http://www.eclipse.org/downloads/packages/eclipse-classic-342/ganymedesr2.

Notez que si vous utilisez *Eclipse* depuis l'IRIT, il est nécessaire de spécifier le proxy dans les Préférences (Menu *Window*, choisissez *Preferences*, section *General*, sous-section *Network Connections*). Reportez-vous au site de l'IRIT pour les valeurs à utiliser [1].

1.2 Installation

L'installation s'effectue depuis le *Software Update* d'*Eclipse* en ajoutant l'*Update Site* de MAY et celui de ses dépendances.

Allez dans le menu *Help* puis choisissez *Software Update*. Allez dans l'onglet *Available Software*.

Ajoutez les *Update Sites* suivants :
– http://www.irit.fr/PERSONNEL/SMAC/noel/may/update/
– http://oawbranch.pluginbuilder.org/releases/p2-updateSite/
Pour ajouter un *Update Site*, cliquez sur *Add Site*.

Dans la liste des *Update Site*, dépliez *Make Agents Yourself* et cochez *Make Agents Yourself*. Cliquez sur *Install*, suivez les instructions, redémarrez *Eclipse* comme proposé.

1.3 Mise-à-jour

Dans le futur, des mises à jour seront disponibles. Pour les effectuer, voici la procédure :

Allez dans le menu *Help* puis choisissez *Software Update*. Allez dans l'onglet *Installed Software* et cliquez sur *Update*. Suivez les instructions, redémarrez *Eclipse* comme proposé.

1. http://intranet.irit.fr/Les-principaux-filtres-d-acces

2 Processus typique de développement d'un agent avec MAY

Comme expliqué précédemment, MAY est un outil permettant de spécifier des μ-architectures à base de μ-composants dans le but de développer une API qui sera redistribuée au développeur de l'application (qui fera abstraction de l'implantation fournie du niveau conteneur).

Pour cela, plusieurs phases de développement sont à dissocier :
- Spécification de μ-composants et de μ-architectures :
  - Créer un projet MAY
  - Décrire le modèle à l'aide de μ-ADL
  - Générer le code correspondant au modèle
- Implantation des μ-composants avec JAVA
- Implantation d'une *Factory* avec JAVA
- Génération de l'API distribuable sous forme de JAR

Les sections suivantes donneront des explications plus précises sur ces phases de développement. Nous verrons les outils que MAY ajoute à *Eclipse*, une description de μ-ADL et un certain nombre de recommandations pour développer un agent et ses μ-composants.

3 Outils ajoutés à *Eclipse*

Les outils proposés par MAY sont les suivants :
- Création d'un projet MAY
- Création de fichiers de description de modèles d'agent « textuels »
- Génération du code correspondant à un modèle d'agent

Dans le futur, nous prévoyons d'ajouter à MAY un éditeur graphique de μ-architectures et μ-composants.

3.1 Création d'un projet

Un projet MAY est simplement un projet *Eclipse* JAVA avec un certain nombre de dépendances, un fichier de description de modèle vide et les répertoires nécessaires créés par défaut.

Pour créer un nouveau projet, allez dans le menu *File*, *New*, *Project...*, catégorie *Make Agents Yourself* et choisissez *MAY Project*. Suivez les instructions.

3.2 Création d'un fichier de description de modèle d'agent « textuel »

Un modèle d'agent est représenté par un fichier texte contenant des spécifications de μ-composants et de μ-architectures décrits avec μ-ADL (présenté section 4). L'extension de ces fichiers est `muadlt`. L'éditeur propose une colorisation syntaxique, une complétion plutôt limitée et une vérification sommaire de la validité du modèle.

Pour créer un nouveau fichier de description de modèle d'agent, allez dans le menu *File*, *New*, *Others...*, catégorie *Make Agents Yourself* et choisissez *MuADL Text file*. Suivez les instructions.

3.3 Génération du code correspondant à un modèle d'agent

Cette action génère les classes nécessaires à l'implantation des μ-composants décrits dans le modèle (les `QuasiComponent`), la classe représentant le médiateurs de la μ-architecture décrite dans le modèle et la classe façade représentant l'agent conforme à la μ-architecture.

Pour générer le code correspondant à un modèle d'agent, faites un clic droit sur un fichier de description de modèle (un fichier `.muadlt`) et dans le sous-menu *Make Agents Yourself*, choisissez *Generate QC and Architecture*.

Le code généré se trouvera dans le répertoire `src-gen` (qui doit donc exister dans le projet où se situe le fichier de description de modèle d'agent).

Il est important de noter ici qu'il ne faut jamais toucher au contenu du répertoire `src-gen` pour la simple et bonne raison qu'il est entièrement vidé à chaque génération du code correspondant à un modèle d'agent !

4 μ-ADL : Langage de description de modèle d'agent

Comme dit précédemment, MAY permet, grâce au langage de description de modèle d'agent μ-ADL, de spécifier :

- Des μ-composants qui :
  - Fournissent un certain nombre de méthodes
  - Requièrent (de la μ-architecture) des méthodes fournies par d'autres μ-composants
  - Requièrent (de la μ-architecture) la possibilité de changer d'autres μ-composants
  - Possèdent un état que la μ-architecture doit transférer lors d'un changement du μ-composant
- Des μ-architectures qui possèdent un certain nombre de μ-composants interdépendant (conformement à leurs requis) séparés en « niveaux »
- Des types de donnés utilisés par les μ-composants correspondant à des classes JAVA

Les « niveaux » de la μ-architecture permettent de séparer conceptuellement la partie « conteneur » du modèle d'agent de la partie « application ». Typiquement, la première contiendra les mécanismes opératoires de l'agent, tandis que la seconde contiendra son ou ses comportements, qui devront donc être spécifiés comme μ-composants.

μ-ADL ne permet pas de spécifier comment seront implanter les μ-composants, c'est-à-dire pas d'information sur les constructeurs, les membres de classe ou les méthodes privées

Nous détaillons ici comment spécifier chacun de ces éléments.

4.1 DataTypes

On fera référence à des classes JAVA en leur donnant un nom à l'intérieur du fichier de description de modèle pour être utilisées dans les définitions de μ-composants.

Les différents syntaxes sont les suivantes :

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

// exemples
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>
```

Pour le moment, aucune vérification n'est faite sur les classes spécifiées dans les DataType.

Il important de comprendre que les DataTypes ne sont que des alias, et que l'on ne peut pas les utiliser comme paramètre de générique.

4.2 μ-composants

Voici les éléments composant un μ-composant :
- Un nom
- Des méthodes fournies (`provided`)
- Des méthodes requises (`required`)
- Des attributs pour lesquels la future μ-architecture devra préserver la persistance lors de mutations (`persistent`)
- Un ensemble de μ-composants que l'agent pourra changer en cours d'exécution (`change`)

Ce point est susceptible de changer dans le futur

Les méthodes de μ-composants sont similaires à celles de JAVA : elles possède un nom, des arguments nommés typés ordonnés, et un type de retour. Pour faire référence à une méthode, il suffira de connaître son nom, le type de ses arguments ordonnés et son type de retour. De plus, comme en JAVA, deux méthodes ayant le même nom et les mêmes types d'arguments ne pourront pas coexister dans un même μ-composant (ni dans une μ-architecture actuellement).

Les méthodes requises sont spécifiées par une référence de méthode qui devra être fournie par un des μ-composants de la μ-architecture.

En pratique, on spécifie pour chaque μ-composant un package pour générer le code.

Un exemple de 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

Une μ-architecture possède :
- Un nom
- Un ensemble de μ-composants dits de niveau conteneur (`container`)
- Un ensemble de μ-composants dits de niveau application (`application`)
- Un ensemble de modificateurs de visibilité sur les méthodes des μ-composants (`visibility`)

Cette manière de spécifier l'accès à l'agent de l'extérieur est susceptible de changer dans le futur

Les μ-composants présents dans les deux niveaux peuvent être spécifiés comme étant changeables (avec `changeable`) par d'autres μ-composants. Seuls des μ-composants d'un même niveau peuvent se changer entre eux.

Par défaut, une méthode de μ-composant du niveau application est visible de tout les μ-composants de la μ-architecture et une méthode de μ-composant du niveau conteneur est visible par les autres μ-composants du niveau conteneur. Ce comportement peut être changé en spécifiant (avec `application`) qu'une méthode d'un μ-composant du niveau conteneur est visible au niveau application.

On peut aussi spécifier qu'une méthode d'un μ-composant du niveau conteneur est accessible depuis l'extérieur de l'agent (avec `external`).

Encore une fois, on spécifie pour chaque μ-composant un package pour générer le code.

Un exemple de μ-architecture :

```
MuArchitecture MyAgent {
  package another.test.package
  container {
```

```
    changeable LifeCycle
    Message
  }
  application {
    Behavior
  }
  visibility {
    external Message.send(String)
    application Message.receive() : String
  }
}
```

Notons donc que :
- Si un μ-composant `A` de la μ-architecture requière une méthode d'un μ-composant `B`, alors `B` devra obligatoirement être présent dans la μ-architecture.
- Si `A` est dans `application` et `B` dans `container`, alors cette méthode requise doit être spécifiée comme visible par `application` dans la partie `visibility` de la μ-architecture.
- Les requis ne sont vérifiés qu'au niveau de la μ-architecture.
- Deux μ-composants ne peuvent pas fournir des méthodes ayant la même signature.

Cela devrait changer dans le futur

4.4 Informations complémentaires

μ-ADL permet l'utilisation de commentaires :

```
// un commentaire
/*
un commentaire sur plusieurs lignes
*/
```

5 Implantation de μ-composants, utilisation de μ-architectures et distribution d'API

Une fois le code correspondant à un modèle généré (voir 3.3), l'implantation des μ-composants et l'utilisation des μ-architectures est possible en JAVA.

Après la génération du code correspondant à un modèle d'agent décrit dans un fichier `.muadlt`, un ensemble de classes se situera dans le répertoire `src-gen`. Comme dit précédemment, le contenu de ce répertoire ne devra pas être modifié. Pour implanter les μ-composants, on écrira des classes JAVA étendant certaines des classes générés comme nous allons le voir. De manière générale, on placera les implantations dans le répertoire `src`.

5.1 Implantation de μ-composants

Pour chaque μ-composant de nom `X` défini dans le modèle, MAY génère une classe `QuasiComponentX` et une classe `AdapterX` La première devra être étendue pour être implantée.

Par exemple, voici deux classes générées possibles pour un cycle de vie :

```
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();
}
```

La première étend la classe `QuasiComponent` qui prend en paramètre la classe `AdapterLifeCycle` et spécifie les méthodes fournient par le μ-composant. `AdapterLifeCycle` représente le type de μ-architecture à laquelle pourra être raccordé ce μ-composant : elle spécifie les méthodes requises par le cycle de vie. Pour chaque μ-architecture utilisant ce μ-composant, une implantation correspondante à `AdapterLifeCycle` sera générée.

Ainsi, les méthodes requises sont directement mises à disposition de l'implantation en faisant appel à l'architecture à l'aide d'une méthode protégée de la classe `QuasiComponent` : `this.architecture()`.

Voici une implantation possible de ce 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;
  }
}
```

Les points importants à remarquer sont :

- Une méthode `void start()` est présente pour exécuter des instructions lors du démarrage de l'agent
- `this.architecture()` permet d'accéder aux méthodes fournies par d'autres μ-composants à travers la μ-architecture

Voici un exemple de μ-composant possédant un état persistant :

```
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 {}
```

Elle étend la classe `QuasiComponent` qui prend en paramètre la classe `AdapterMessage` et implémente l'interfaces `WithPersistentState<PersistentStateMessage>`.

`WithPersistentState<PersistentStateMessage>` spécifie les méthodes nécessaires à la gestion de l'état persistant du μ-composant (qui seront appelées lors d'un changement du μ-composant).

Voici une implantation possible de ce 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) {
      // TODO Auto-generated catch block
      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) {
      // TODO Auto-generated catch block
      e.printStackTrace();
      return "";
    }
  }

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

Remarquons que l'on utilise un objet de type `PersistentStateMessage` pour stocker et lire l'état persistant.

Voici un exemple de μ-composant changeant un autre μ-composant :

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

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

Notons qu'une méthode `void change(QuasiComponentMessage _comp)` a fait son apparition dans `AdapterAdaptation`. Celle-ci permettra à l'implantation d'accéder à la fonctionnalité de changement de μ-composant.

Voici une implantation possible de ce μ-composant :

```
public class Adaptation extends QuasiComponentAdaptation {

  @Override
  public void start() {}

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

Nous voyons aussi pour la première fois l'instanciation d'une implantation d'un μ-composant.

Lors de l'appel de `change(...)`, la μ-architecture récupérera automatiquement l'état de `Message` puis l'injectera dans le μ-composant nouvellement instancié à l'aide des méthodes définies dans `Message` vu précédemment.

5.2 Utilisation de μ-architecture

L'utilisation d'une μ-architecture revient à créer un agent conforme à cette μ-architecture en instanciant ses μ-composants.

Pour cela, le code généré d'une μ-architecture `X` comprend deux classes importantes : `XMediator` et `X`. La première représente la μ-architecture elle-même, et la seconde l'agent lui-même faisant office de façade proposant les méthodes externes.

Ceci est susceptible de changer dans le futur

Ainsi, créer un agent conforme à une μ-architecture `SimpleAgent` possédant 4 μ-composants `Adaptation`, `Message`, `LifeCycle` et `Behavior` se fait de la manière suivante :

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

Notons en particulier l'appel de la méthode `start()` qui démarre chacun des μ-composants de la μ-architecture .

Actuellement, l'ordre d'appel de ces méthodes n'est pas déterminé

5.3 *Factory*

Un des objectif de MAY étant de diminuer l'effort de développement en distribuant des modèles d'agents « à la carte » prêt à l'emploi, il est nécessaire de pouvoir « pré-remplir » les μ-architectures pour permettre l'instanciation directe de celles-ci par un utilisateur.

Cela se fait à travers l'utilisation d'une *Factory* qui, pour le moment, doit être programmée à la main. Dans le futur, cette *Factory* sera générée automatiquement à partir d'une description d'une instanciation partielle de la μ-architecture.

Créer une *Factory* revient à créer une classe proposant une (ou plusieurs) méthode statique instanciant des μ-composants et un agent. Par exemple :

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

Ainsi, la création d'un agent peut se faire depuis n'importe quel programme en faisant appel à : `MySimpleAgent.create(...)`.

Par exemple :

```
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 Redistribution de l'API

Les classes créées dans la section précédente représentent une API utilisable par un utilisateur du modèle d'agent pour créer des agents.

Redistribuer ceux-ci revient à exporter le projet sous forme de JAR à l'aide d'*Eclipse*. Pour cela, faites un clic droit sur le projet MAY, choisissez *Export...*, dans la catégorie *Java*, choisissez *JAR File* et suivez les instructions.

Le fichier JAR résultant devra être distribué avec le fichier `fr.irit.smac.muadl.impl_X.X.X.jar` disponible sur le site de MAY. Ils pourront tout les deux être ajoutés comme dépendances à un projet *Eclipse*.

6 Retours

Des problèmes peuvent survenir à différents niveaux :
– Lors de l'utilisation de μ-ADL pour spécifier des modèles :

  - Erreurs qui devraient apparaître et qui n'apparaissent pas
  - Erreurs qui ne devraient pas apparaître et qui apparaissent
  - Concepts non modelisables
- Lors de la génération de code :
  - Code non généré
  - Code généré qui contient des erreurs JAVA
  - Code généré qui ne correspond pas aux spécifications

Dans le cas de code non généré, une trace de la génération est disponible dans la console depuis laquelle *Eclipse* a été lancé.

Merci de me remonter tout ça, accompagné d'exemples qui font apparaître le problème !

Si il y a des remarques sur le tutoriel, n'hésitez pas non plus.