Politecnico di Milano
Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Design and development of a new editor for the `SelfLet` environment: `SelfLetClipse2`

Relatore: Elisabetta DI NITTO
Correlatore: Nicolò Maria CALCAVECCHIA

Tesi di Laurea di: Ngoc Hoang PHAM
Matr. 722839

Anno Accademico 2010-2011

To my parents and my dearest friend Binh Thai

Ringraziamenti

Thanks to my tutor Nicolo Calcavecchia for his suggestions and guidances during the course of the thesis. Thanks to professor Elisabetta Di Nitto for her suggestions and contributions to the thesis.

Thanks to my dearest friend Binh Thai for her understandings, patience and support while I was doing the thesis.

Thanks to the Eclipse Graphical Modeling Framework Community Forum Members for their kindness in helping newcomers into mastering the framework.

Milano, November 27th 2011

Ngoc Hoang PHAM

Table of Contents

Chapter 1

Introduction

With the tremendous evolution of the internet, the emergence of many application models such as the Cloud computing, Software as a Service, Infrastructure as a Service, and the expansion of varying business domain models, the computer systems' complexity has approached the limits of human capability. Autonomic computing, inspired by a term in biology that is the autonomous nervous system in the human body, is a specific field of research with the hope to find an answer to this complexity problem. It was initially introduced by IBM researchers in 2001 and thereafter in 2003 [19] with their widely influential paper: The vision of autonomic computing. In this paper, Kephart and Chess described a system with self management capabilities representing by four aspects: self-configuration, self-optimization, self-healing and self-protection; the intent is to free the system administrators from the burdening of configuration, operation and maintenance while still guarantee the maximum performance in a continous manner.

There have been many efforts in the industry in designing, building such self managing systems[18, 22]. One of the approach that was taken at Politecnico di Milano is the `SelfLets` approach [1, 4], a `SelfLet` is an Autonomic Element that can: accomplish its goals by running well specified behaviors, interact with other running `SelfLets` to obtain the help in achievement of its goal, monitoring itself and its neighbors to discover potential problems or challenges for improvement and applying autonomic policies. In the original paper of the `SelfLet` , Bindelli et al. have described the `SelfLet` model and showed an example of a `SelfLet` with some preliminary results. In the following development papers made by Calcavecchia et al. [4], they have proposed a prediction model for the `SelfLet` along with an Integrated Development Environment for `SelfLet` called `SelfLetClipse`. This implementation of the IDE has dependencies to the ArgoUML libraries[8]. The work in this thesis focuses on the development of the new editor for `SelfLet` framework called `SelfLetClipse2` utilizing the powerful Eclipse Modeling Framework (EMF), Graphical Editing Framework (GEF) and Graphical Modeling Framework (GMF) in order to create the stunning graphical interfaces for diagrams of implementation behaviors of `SelfLets`' services at the same time persist the data model underneath the generated diagram. The result is a fully integrated development environment in Eclipse for the `SelfLet` framework that is easy to create, maintain and develop.

The remaining of the thesis is organized as following: the second Chapter briefly describes the `SelfLet` model, its development lifecycle and the currently available `SelfLetClipse`, the third Chapter presents the new features of the new editor `SelfLetClipse2`, the fourth Chapter detailed the meta-model for the graphical editor and its corresponding EMF Editor, the next Chapter goes deeper into the implementation in which the combination of the three frameworks: EMF, GEF, and GMF is employed in order to create the desired graphical interfaces, the sixth Chapter comes with examples of creating a complete `SelfLet` project, then the conclusion Chapter draws some important points for future improvements.

Chapter 2

The `SelfLet` model

2.1 Introduction

The essence of autonomic computing is the capabilities of self management, any implementation of autonomic computing should take into consideration the four principal aspects of autonomic computing:

Self-configuration : the idea is to reduce at minimum the amount of manual work done by humans, components and systems are self configured with only the presence of high level policies.

Self-optimization : components and systems seek to improve themselves with respect to performance and efficiency.

Self-healing : the system is capable of detecting failures by itself and repairs itself without the interception of the system administrators.

Self-protection : the system automatically defends itself against the attacks or cascading failures.

These principles have been applied into the construction of the `SelfLet`. Specifically, each `SelfLet` is a self-sufficient piece of software which is located in some kinds of logical or physical network. Multiple `SelfLets` are organized using a network topology, for example with a direct connection between two `SelfLets` , so that they can interact and communicate with each others as in Figure 2.1.

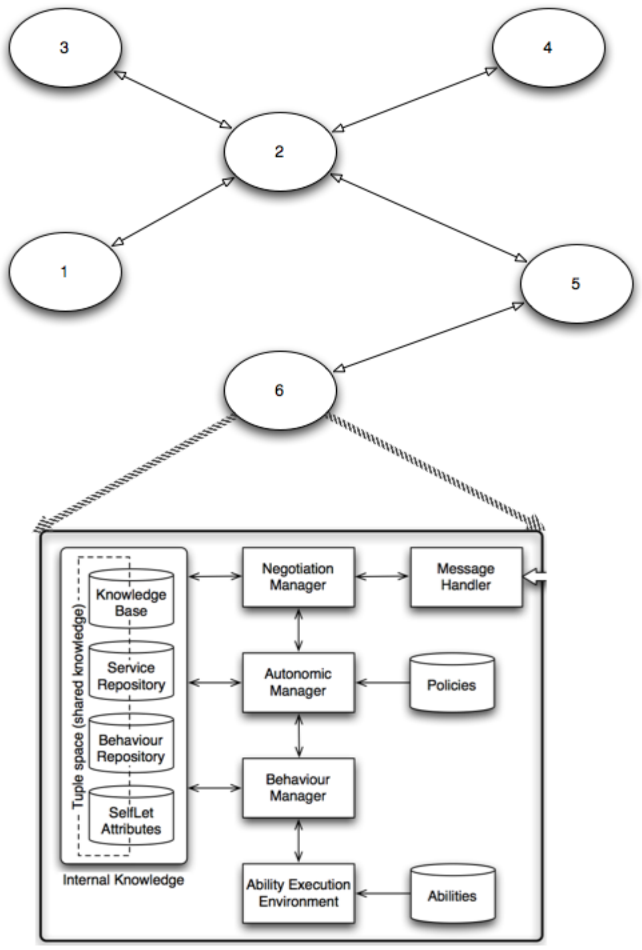


Figure 2.1: The network of SelfLets interacting with each other

2.2 The internal structure of the SelfLet

As we can see in Figure 2.1, the SelfLet contains the Internal Knowledge, the Negotiation Manager, the Autonomic Manager, the Behavior Manager and Ability Execution Environment. All of these elements reflect the four aspects of the self-management autonomic system.

- *Internal Knowledge*: contains the SelfLet's Knowledge Base, the Service Repository, the Behavior Repository, the SelfLet 's attributes. Service Repository and Behavior Repository are used to store the Services and Behaviors that the SelfLet can do or can teach as described above.

- *Negotiation Manager*: manages the communication with other SelfLets for the acquisition of Services and Behaviors Implementations.

- *Autonomic Manager*: manages the Policies existing in the `SelfLet` , it monitors the events and fire the corresponding rules associated so that `SelfLet` can automatically improve itself efficiently.
- *Behavior Manager*: runs the Behavior implementations within the `SelfLet` , at one point in time, there might be two or more current behavior implementations running due to the nested behavior implementation or the waiting between remote behaviors' requests.
- *Ability Execution Environment*: installs, uninstalls, stores and executes abilities in .jar file extension

2.3 The `SelfLet` Conceptual Model

The conceptual model of the `SelfLet` is reported in Figure 2.2 with the main element is `SelfLet` uniquely identified by an ID. *SelfletProperties* and *SelfletResources* form the two main elements of a `SelfLet`.

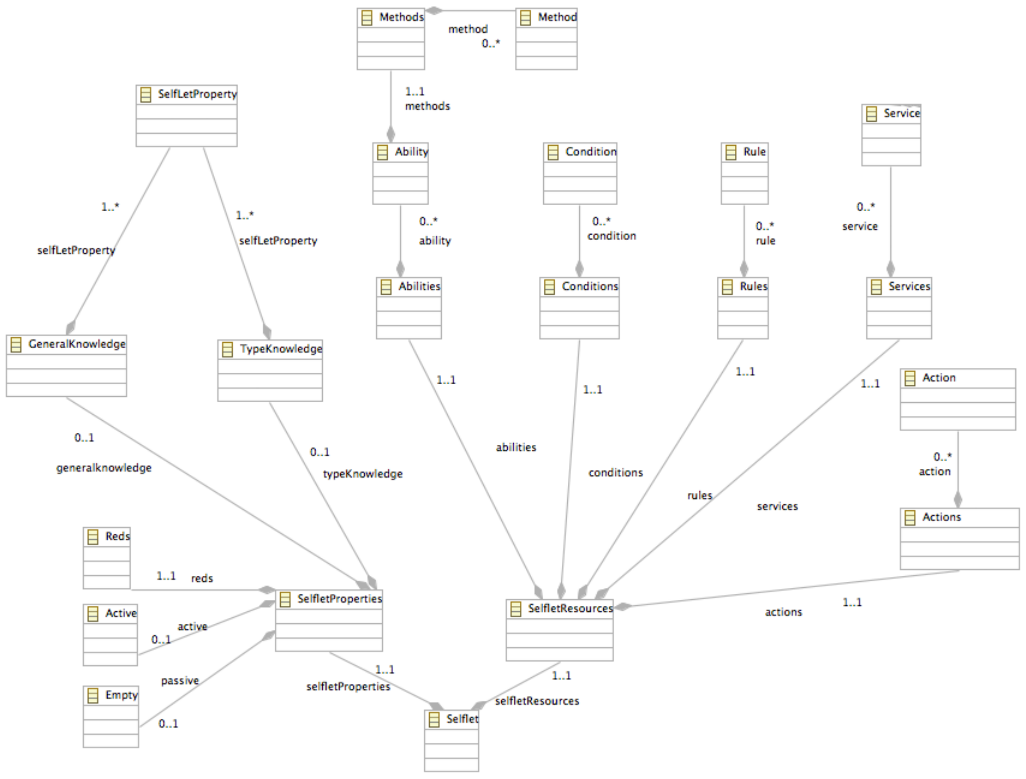


Figure 2.2: The conceptual model of a `SelfLet`

- *SelfletProperties*: describes the `SelfLet` itself with information about the location, the type of the `SelfLet` and the type of knowledges it has along with values.
  - *General Knowledge*: The general knowledge as it says in the name for the `SelfLet`, it might be the name of the creator, or anything.
  - *Type Knowledge*: The type knowledge is used to characterize the `SelfLet` with distintive properties.
  - *Reds*: The IP Address and Port of the current location of the `SelfLet`
  - *Active/Passive*: Specify if the current `SelfLet` is Active or Passive, and the main Service that `SelfLet` is trying to achieve.
- *SelfletResources*: contains all the necessary resources that a `SelfLet` needs to operate in a self-managed manner.
  - *Services*: is the central of the `SelfLet`, *Service* specifies high level description of a task. A *Service* has input parameters and at the end of the execution produces the result output, *Services* can also be exchanged among *SelfLets* through its *OfferMode* property. The model for the *Service* is too complex to put into the same diagram of the `SelfLet` so it is separated into Figure 2.3. As we can see in Figure 2.3 a `SelfLet` can have many behavior implementations. Behavior is an abstract class that are implemented by Elementary Behavior or Complex Behavior.
  - *Behaviors*: Behaviors are implemented as state diagrams with many states and transitions between States, Initial States and Final States are used in both Elementary Behavior and Complex Behavior, while Intermediate State is used only in Complex Behavior Diagram, and Invocation State is used only in Elementary Behavior Diagram.
    * *Elementary Behavior*: represents a low level objective that is directly implemented by an ability, this is the concrete implementation of a *Service*, and it is also the simplest implementation of a *Service*, it starts with an Initial State, then an Ability State and then a Final State. These States are connected together.

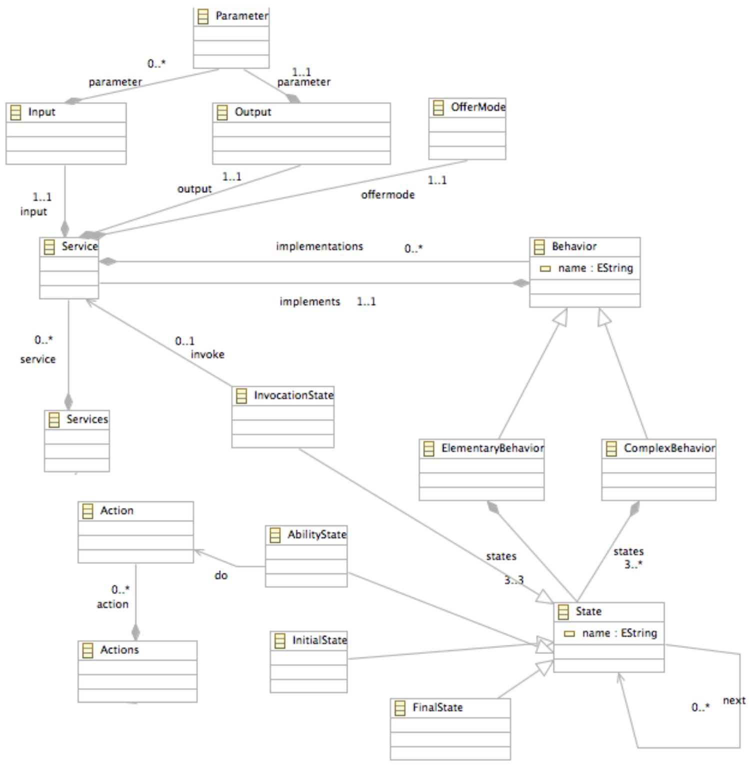


Figure 2.3: The model of Services in the `SelfLet`

- *Initial State*: is the node of the beginning of every Behavior implementation. It does not contain any state information.
- *Invocation State*: The Invocation State can do some *Actions* and execute the Ability file with .jar extension.
- *Final State*: is the node that terminate the Behavior implementation. After the Final State, the Service may produce its result as the output specified when it is declared.

∗ *Complex Behavior*: represents a high level objective and typically contains different subgoals, this is the concrete and complex implementation of a *Service*, it starts with an Initial State and then any of the Intermediate State, these Intermediate States are free to connect to other States, however, there must be a path from the Initial State to go to the Final State in the diagram.

- *Initial State*: is the node of the beginning of every Behavior implementation. It is also a normal State.
- *Intermediate State*: The Intermediate State can do some

    Action and invoke other Services in case it needs help from the neighbors. Complex Behaviors can contain one or many Intermediate States, these States may connect to each others or go to the Final States.

    · *Final State*: is the node that terminate the Behavior implementation. After the Final State, the Service may produce its result as the output specified when it is declared. It is also a normal State.

  – *Actions*: specify the task that a *State* in the Elementary Behavior or Complex Behavior can take when the `SelfLet` enter to that State.

  – *Conditions*: contain the condition that is written in XML format, that specify the condition on transitioning between states in the Behaviors' State diagram.

  – *Abilities*: contain the executable program unit that is normally written in Java and packaged in a .jar packages, this program unit performs a specific task in the `SelfLet`. This Ability file is invoked by one State in the Elementary Behaviors. Later in the Chapter , we will see in more details the Elementary Behaviors.

  – *Rules*: (or *Policies*) specify how the system reacts to events that happen. Rules are written using an open source business rule management system called Drools[7]. Rules can be used to enable or disable a *Service*

`SelfLet`'s *Service* can be offered in one of the following modes:

- *Can Do*: The *Service* itself can do the ability through one of its Elementary Behavior and return the values to the caller.

- *Can Teach*: The *Service* can teach the caller on how to do the specific Behavior, by replicating one of its implementation Behaviors.

- *Can Do and Can Teach*: The *Service* can even do and teach

- *Know Who Can Do*: The *Service* knows who can do the certain Behaviors, it keeps a list of other Services in its knowledge base.

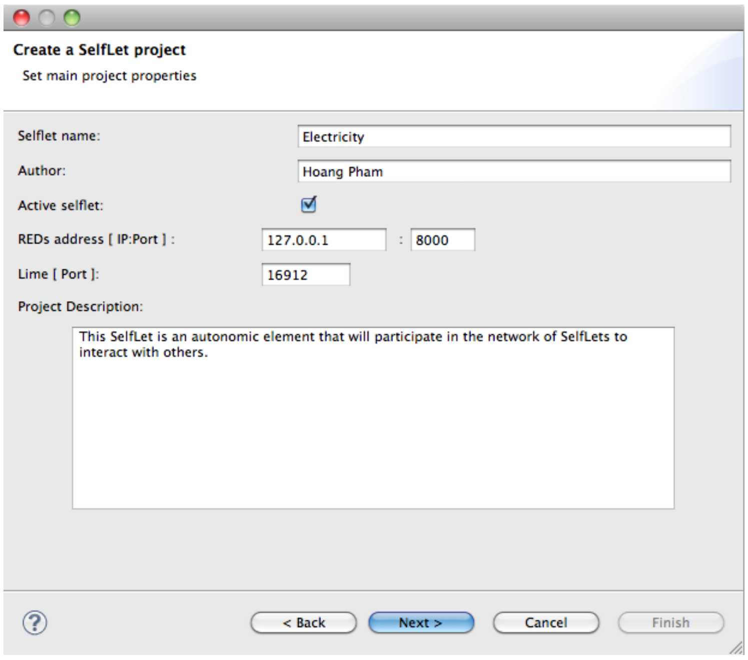
- *Know Who Can Teach*: The *Service* knows who can teach the Behaviors, it also keeps a list of other Services in its knowledge base.
- *Know Who Can Do and Teach*: The *Service* knows who can both do and teach, it keeps a list of other Behaviors in its knowledge base.
- *None*: The *Service* can not do or teach anything, neither know who can do nor who can teach.

2.4 The current implementation of SelfLet: SelfLetClipse

SelfLet Integrated Development Environment has been previously implemented as an Eclipse Plugin with the use of ArgoEclipse by Nicola Calcavecchia: SelfLetClipse[4]. This implementation focuses on the use of Eclipse Graphical Modeling Framework to make graphical diagrams, integrates the current version of SelfLetClipse but removes all the dependencies from ArgoEclipse.

SelfLetClipse is a plugin that supports 3 wizards for the creation of SelfLet project:

- SelfLet: this wizard creates the SelfLet project with the following folders: Abilities, Actions, Behaviors, Goals, Rules, Conditions and a SelfLet configuration file. Figure 2.4 shows the first step in creating a SelfLet project, it first asks the developers the information about the SelfLet itself, the properties and its general knowledge. Figure 2.5 shows the second step in creating a SelfLet project, it asks the developers the main Goal that the SelfLet is trying to achieve with the input and output parameters, the output parameter is specified by using a checkbox, the input parameters can be more than one. Figure 2.6 shows a screenshot of the current version of the SelfLetClipse IDE project folder structure created and the default behavior of the SelfLet by drawing an ArgoEclipse diagram with state information.
- *Selflet Goal*: this wizard is used to add a new Goal into the SelfLet project, Goal is represented as an XML file with name and input, output

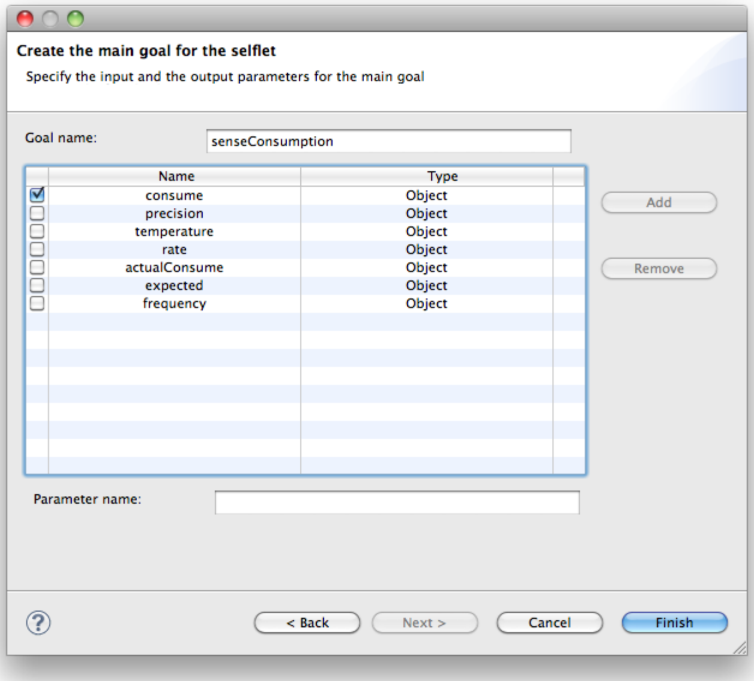


Create a SelfLet project

Set main project properties

Selflet name:	Electricity
Author:	Hoang Pham
Active selflet:	✓
REDs address [IP:Port] :	127.0.0.1 : 8000
Lime [Port]:	16912
Project Description:	This SelfLet is an autonomic element that will participate in the network of SelfLets to interact with others.

Figure 2.4: The `SelfLetClipse`'s `SelfLet` wizard properties description



Create the main goal for the selflet

Specify the input and the output parameters for the main goal

Goal name: senseConsumption

	Name	Type
✓	consume	Object
☐	precision	Object
☐	temperature	Object
☐	rate	Object
☐	actualConsume	Object
☐	expected	Object
☐	frequency	Object

Parameter name:

Figure 2.5: The `SelfLetClipse` *Goal* description

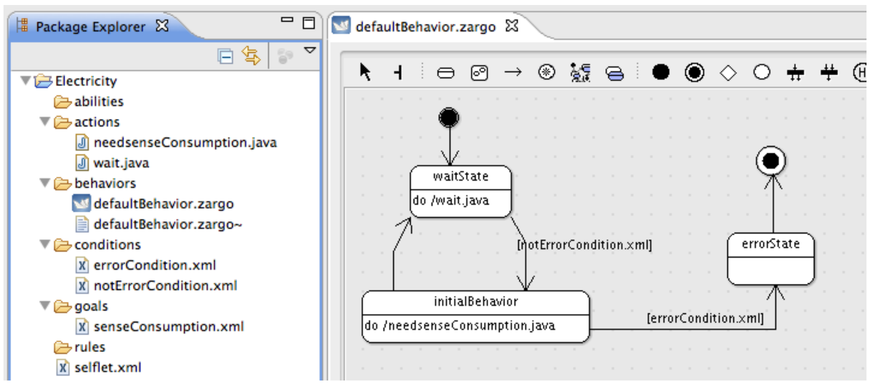


Figure 2.6: The `SelfLetClipse` project structure and default behavior implementation

parameters. `SelfLet` developers have to create the behavior implementations by themselves.

- *Selflet Behavior*: this wizard is used to add a behavior implementation into the `SelfLet` project by choosing which Goal it is implementing.

The work carried out by this implementation by Calcavecchia [4, 2, 5] relies heavily on the uses of ArgoEclipse libraries to draw the Elementary/Complex Behavior diagrams. Our work tries to remove these dependencies and builds from scratch a new graphical editor using the Eclipse Graphical Modeling Framework in combination with the Eclipse Modeling Framework and Eclipse Graphical Editing Framework as we will later see in Chapter 5.

2.5 Conclusion

In this Chapter, we have reviewed the conceptual model of the `SelfLet` along with its most complex structure, the *Service*, we also briefed the current implementation of `SelfLet`, the `SelfLetClipse`. Based on these conceptual models and the existing implementation, we are able to design and develop the graphical interfaces representing the *Service* in terms of state diagrams, UML representations and improving the current `SelfLetClipse` IDE with the more in depth information about the `SelfLet`.

In the next Chapter, we will see the new features of the editor, we call it `SelfLetClipse2`, and how it can be used to assist the `SelfLet` developers in designing their desired autonomic systems.

Chapter 3

The main features of the new Editor: `SelfLetClipse2`

3.1 Introduction

In this Chapter, we discuss in details the features and functionalities of the new editor `SelfLetClipse2` . `SelfLetClipse2` is based on the standard Eclipse platform, so it enjoys all the features and functionalities of an Eclipse plugin, besides that, it combines three popular and industry proven frameworks Eclipse Modeling Framework (EMF)[12], Eclipse Graphical Editing Framework (GEF)[9], Eclipse Graphical Modeling Framework (GMF)[13] in order to create the remarkable graphical editor.

Eclipse Modeling Framework EMF : EMF is a modeling framework and code generation facility for building tools and applications based on a structured data model. This structured data model is normally written as Ecore Model. The framework then creates a helper generator model based on this Ecore Model and generates the necessary classes and an editor for testing the model.

Eclipse Graphical Editing Framework GEF : GEF provides technology to create rich graphical editors and views for the Eclipse Workbench User Interfaces.

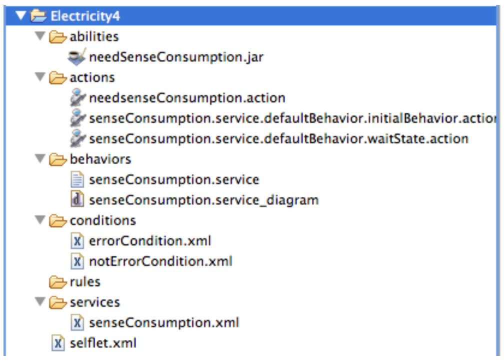
Eclipse Graphical Modeling Framework GMF : GMF bridges the gap between the EMF and GEF to enable the developers to build graphical editor based on their model written using EMF.

3.2 The new graphical editor `SelfLetClipse2`

In the new implementation of the `SelfLetClipse2`, we reuse the `SelfLet` wizard of the previously available `SelfLetClipse` IDE to let the developers create the standard `SelfLet` project. However, at the end of the wizard, the actions taken are modified in order to create the default diagram and default diagram data model. The two wizards `SelfLet` Goal and `SelfLet` Behavior of the previous `SelfLetClipse` are changed to `SelfLet` Service and `SelfLet` Service Diagram respectively. The new structure of the `SelfLetClipse2` wizards is as follows:

- `SelfLet`: This wizard is used to create the `SelfLet` project, in the first step of the wizard, the `SelfLet` developers specify the `SelfLet` properties the same as in the previous implementation of `SelfLetClipse` in the next step, then the main Service that the `SelfLet` is trying to achieve, this step has the same interface as the previous `SelfLetClipse`, however, the term Goal is now changed into Service, developers have to insert the name of the Service along with its input and output parameters, there can be zero or many input parameters but there must be one output parameter for a Service. At the end of the wizard, the `SelfLet` project is created in the workspace along with the default Service diagram and the default Service data model as in Figure 3.1. This Service diagram and Service data model are based on the GMF framework instead of the ArgoUML as in the previous implementation. The Service diagram created includes the default behavior implementation for the Service as shown in Figure 3.2

- `SelfLet` Service: Once we have the `SelfLet` project in the workspace, we can add more Services into the project by using this wizard. When the wizard is started, the developers also have to declare the Service name and its input and output parameters as when they create the `SelfLet`



```
▼ Electricity4
  ▼ abilities
      needSenseConsumption.jar
  ▼ actions
      needsenseConsumption.action
      senseConsumption.service.defaultBehavior.initialBehavior.actio
      senseConsumption.service.defaultBehavior.waitState.action
  ▼ behaviors
      senseConsumption.service
      senseConsumption.service_diagram
  ▼ conditions
      errorCondition.xml
      notErrorCondition.xml
    rules
  ▼ services
      senseConsumption.xml
    selflet.xml
```

Figure 3.1: The `SelfLetClipse2` project structure created

project with the main Service. At the end of this wizard, there are three files created in the project workspace:

serviceName.xml : This file is created in the services folder of the project workspace, it contains the service description and its input and output parameters in XML format.

serviceName.service_diagram : This file contains the graphical diagram for the Service's behavior implementation, it is created in the behaviors folder in the project workspace. The default Service is created with the same name as the Service domain data model. A Service diagram contains behaviors implementations of a Service, each Behavior can be Elementary Behavior or Complex Behavior. Developers use the Tool palette which contains the creation tools for Service's behavior implementation graphical elements to draw into the canvas. The data underneath is saved into the file serviceName.service as described next.

serviceName.service : This file is the underneath domain data model for the diagram, it is the xml file that is created automatically when developers add more elements into the drawing canvas, developers can change the values of this file and the change is reflected immediately in the diagram file. As an example, if the developers do not want to keep the default name for the Service created, they can modify the content of this file and the change is updated into the graph automatically. It is also located in the behaviors folder in the project workspace.

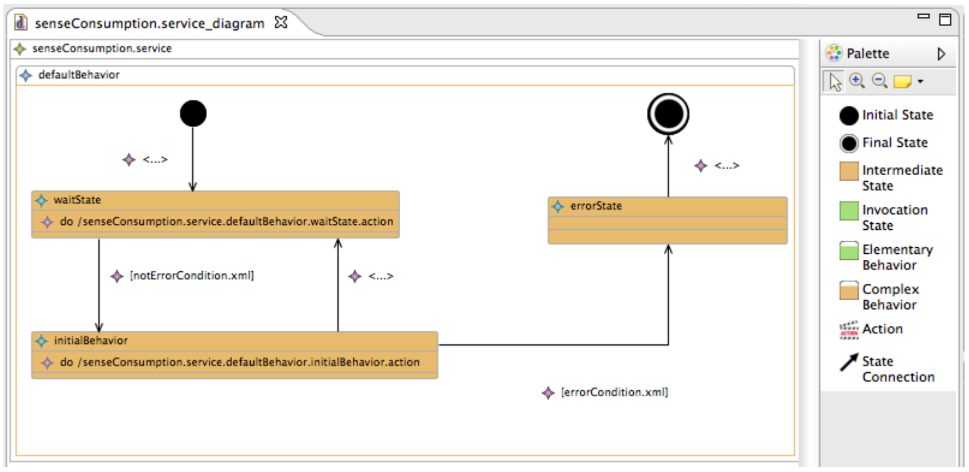


Figure 3.2: The `SelfLetClipse2` default behavior implementation of the Service and the palette

- `SelfLet` Service Diagram: This wizard is used to add additional implementation behavior for a Service that is declared within the `SelfLet` or in other `SelfLets` At the end of the wizard, there are two files created: the serviceName.service and serviceName.service_diagram. The description of these two files are mentioned above.

3.3 The Palette

When the developers create a Service Diagram, a default Service is added into the drawing area, `SelfLet` developers do not need to add more than one Service other than the default Service created, so as a result, there is no creation tool for Service in the graphical editor. The diagram palette is shown in Figure 3.3. In the following section, we describe in details each creation tool of the palette. The order in which the creation tool is described is changed to help explaining the purpose of the Service Diagram. When the Service diagram is first created, it has inside a Service graphical element that is a rectangle and a label, it is created with default size big enough to contain the Elementary Behavior and Complex Behavior.

Elementary Behavior : This creation tool is used to add the Elementary Behavior graphical element into the drawing area of the Service diagram inside the Service rectangle and label. The Elementary Behavior graphical element has vertical layout. One Service can have as many

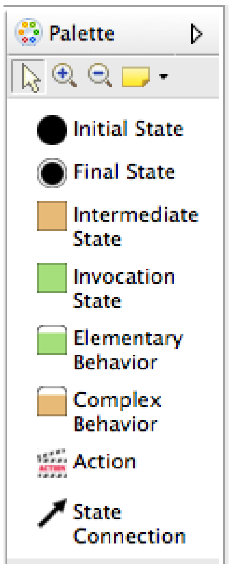


Figure 3.3: The `SelfLetClipse2` Diagram palette

Elementary Behavior graphical elements as desired. When adding new element, the new one is automatically laid out vertically. Elementary Behavior element represents a low level objective that is implemented by an ability so it contains inside only the Initial State, Invocation State and Final State.

Complex Behavior : This creation tool is used to add the Complex Behavior graphical element into the drawing area of the Service diagram inside the Service rectangle and label. The Complex Behavior graphical element has also vertical layout, it is laid along with other Complex Behavior or Elementary Behavior graphical elements. One Service can have as many Complex Behavior graphical elements as desired. Complex Behavior represents high level objective and typically refers to different Services, it can contain inside one Initial State, one or many Intermediate States and one or many Final States. Specific color is used to differentiate between Elementary Behavior graphical element and Complex Behavior graphical element.

Initial State : This creation tool is used to add the Initial State into the Elementary Behavior or Complex Behavior in the diagram, developers can not draw the States elements outside of Behavior graphical elements, neither inside the Service nor out of the Service rectangle. This implementation helps prevent developers from creating unwanted diagrams. Each Elementary Behavior and Complex Behavior must have one Initial State.

Final State : This creation tool is used to add the Final State into the Elementary Behavior or Complex Behavior, the same restriction applies for Final State, however, developers could create more than one Final State in the Behavior graphical element.

Invocation State : This creation tool is used to add the Invocation State to only the Elementary Behavior, as later in the Chapter we will see in more details the Action in Invocation State. A special color is assigned to Invocation State graphical element so that developers know they are creating a State that is different from the Intermediate State as shown below. We have used the features of GMF framework to restrict `SelfLet` developers from putting Invocation State into Complex Behavior.

Intermediate State : This creation tool is used to add the Intermediate State to only the Complex Behavior. The Action available for this State is different from the Invocation State so we differentiate it with Invocation State by a different color.

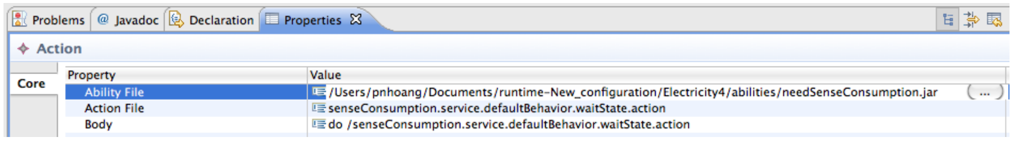
Action : This creation tool is used to add the Action into the Invocation State or Intermediate State, Action for Invocation State differs from Action for Intermediate State. When adding the Action into the Invocation or Intermediate State, the developers can open the Properties view of the Eclipse platform and change the corresponding attributes of the Action. Later in the Chapter, we will see how to do this.

State Connection : This creation tool is used to connect between one State and the others. To add a connection, developers choose this creation tool and drag it from one State to the other. For the diagram to be concise, all the States in the Elementary Behavior or Complex Behavior must have a path that goes from Initial State to Final State. This is implemented in GMF as Audit control, we will see it in greater details in the implementation Chapter 5.

3.4 The Properties view

The advantage of using the Eclipse platform framework is that we can use the underlying architecture for various purpose, in this implementation of the `SelfLetClipse2` we have chosen to override the Properties view in order to achieve the desired behaviors in the system. In this Section, we briefly see which notable properties of each graphical element we can change and how they change the corresponding graphical elements, the project workspace structure and the domain data model underneath.

The Properties view is open by selecting menu Window, Show View, Other, Properties of the standard Eclipse platform. Figure 3.4 show an example of Properties for the Action of the Invocation State in the Elementary Behavior implementation of a Service.



Property	Value
Ability File	/Users/pnhoang/Documents/runtime-New_configuration/Electricity4/abilities/needSenseConsumption.jar
Action File	senseConsumption.service.defaultBehavior.waitState.action
Body	do /senseConsumption.service.defaultBehavior.waitState.action

Figure 3.4: The `SelfLetClipse2` Properties view for Action in Elementary Behavior's State

- *Service*: `SelfLet` developers can change the Service name as differently from the one created by the wizard by navigating to the Properties view. Changing the name reflects immediately in the diagram and also in the domain model. The same applies for Elementary Behavior, Complex Behavior, Invocation State and Intermediate State.

- *Action*: Action has the following properties:

  Ability File : This property is overridden as the customization for the GMF Framework in order to create a File Selector button, clicking on the File Selector button opens a dialog box for selecting the Ability File, it is restricted to only .jar file extension. The selected .jar file is copied into the abilities folder and it serves as the purpose of the Ability for the Invocation State in the Elementary Behavior implementation of the Service.

Action File : The Action file is created automatically when an Action is added into the Intermediate State or Invocation State with the following file name format: serviceName.behaviorName.stateName.action, because it is uniqued for a state in a behavior in a service. Audit controls are used to check if there is a duplicate serviceName or behaviorName or stateName within each levels of containment. The Action file contains the default template for the corresponding Intermediate State or Invocation State in order to guide the `SelfLet` developers into developing further actions as necessary.

Action Body : The Action Body is a string field that stores the visually displayed information of the Action File, it is formatted as: do /ActionFile

- *State Connection*: State Connection has the following properties:

Condition File : This property is overridden as the customization of the GMF framework in order to create a File Selector for the XML condition file, choosing the condition file from the File Dialog will copy it into the project workspace under the conditions folder. The path to the condition file is stored in this field for reference.

Body : This body is automatically created as closed square brackets [Condition File], `SelfLet` developers can also change it to an equation string for example: p = 0.5 for displaying purpose.

3.5 The `SelfLet` development lifecycle

The development of a `SelfLet` system requires an in depth analysis into the application being developed, the objectives and characteristics. The `SelfLet` project is then created using the wizard `SelfLet`, developers have to specify its corresponding properties and a main service with name and input and output parameters. The wizard then creates a sample `SelfLet` project in the workspace with default behavior implementation, the error condition and not error condition file for transition between states in the default behavior of the main service's implementation. A service description file is created with the parameters specified in the wizard. A `SelfLet` XML description file for the whole system is created in the root project workspace. Developer can then add new Service using the `SelfLet` Service wizard or add new Service Diagram using the `SelfLet` Service Diagram Using the palette, developers are able to draw the state diagram representing the behavior implementation of a Service, it can be both Elementary or Complex behavior. By specifying State's Action and State's Connection, developers are building gradually the autonomic `SelfLet` system with Abilities, Actions, Behaviors and Conditions and Service Calls for the `SelfLet` Rules might be added to the rules folder in the project workspace using Drools language [7]. `SelfLet` workspace project files and folders are created in parallel with the file system, at the end of the development, `SelfLet` project can be packaged into a zip folder and it then participates in a network of `SelfLet` autonomously.

3.6 Conclusion

In this Chapter, we have introduced the new features of the `SelfLet` its new wizards, the diagram, the palette and its properties view with several customization. The `SelfLet` development lifecycle briefly describes how the new `SelfLetClipse2` editor is used to help `SelfLet` developers to build their desired autonomic system. In the next Chapter, we will see the `SelfLet` metamodel and the corresponding EMF Editor. Chapter 5 will go in further details into the implementation, how we combined the three frameworks: EMF, GEF and GMF into creating an extraordinary graphical editor for `SelfLet` environment.

Chapter 4

The meta-model and the EMF Editor

4.1 Introduction

In this Chapter, we take a look into the meta-model of the graphical editor that we are going to describe in Chapter 5 and the generated EMF Editor from the Model. We have used the GMF Framework to build the graphical editor, the screenshot of the dashboard is shown in Figure 4.1. As we can see from the Figure, the first step in creating a graphical editor is to create the meta model: the Ecore Model. Ecore model can be created from the following

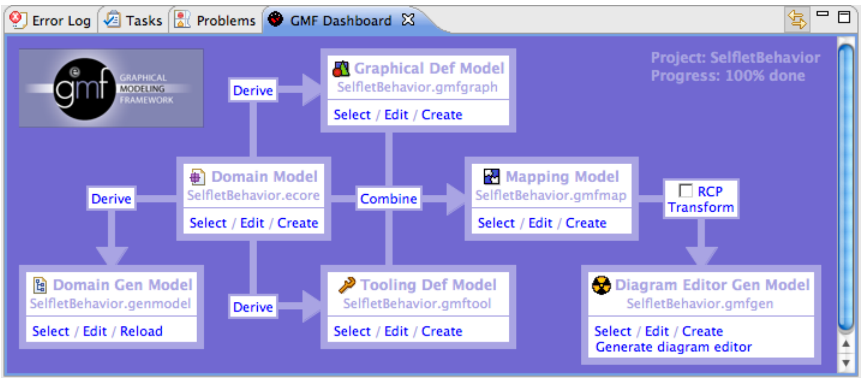


Figure 4.1: The GMF Dashboard

sources:

Annotated Java : The annotated java classes with annotation, such as @model or @containment above the classes declaration, then the EMF compiler can understand these classes as the model for the Ecore meta-model. The EMF book[21] has many in depth tutorials on how to create annotated java classes.

Ecore Diagram : the diagram that contains all the necessary packages, classes, attributes and relationships between classes, this is the most trivial way to build the meta model.

In this implementation, we have chosen to draw directly into the diagram of Ecore our meta model for the editor as we already knew the conceptual model of the `SelfLet` and the `SelfLet`'s most complex structure: Service. The diagram for our Ecore Model (.ecore_diagram) is shown in Figure 4.2, the Ecore model (.ecore) is shown in Figure 4.3. These diagrams are slightly different from the conceptual model we discussed in Chapter 2 because of the simplicity of the graphical editors we implemented.

4.2 The meta-model

We have chosen to directly draw our Ecore model for our graphical editor, let's see in details the components that make up the Ecore kernel model:

EClass : EClass models classes themselves. Classes are identified by name and have a number of attributes and references.

EAttribute : EAttribute models the attributes, EAttributes are identified by name and have a type.

EDataType : EDataType represents simple data types whose details are not modeled as classes, instead they are associated with a primitive or object type that is defined in java. It is identified by name.

EReference : EReference models the associations between classes. It can be aggregation or association or generalization type.

The graphical editor contains diagram of a `SelfLets`' Service, so first we draw a `SelfLet` EClass in the package, this `SelfLet` contains an Aggregation connection to Service EClass so later on in the mapping, we can define the graphical representation for the Service and map the Service to a tool in the

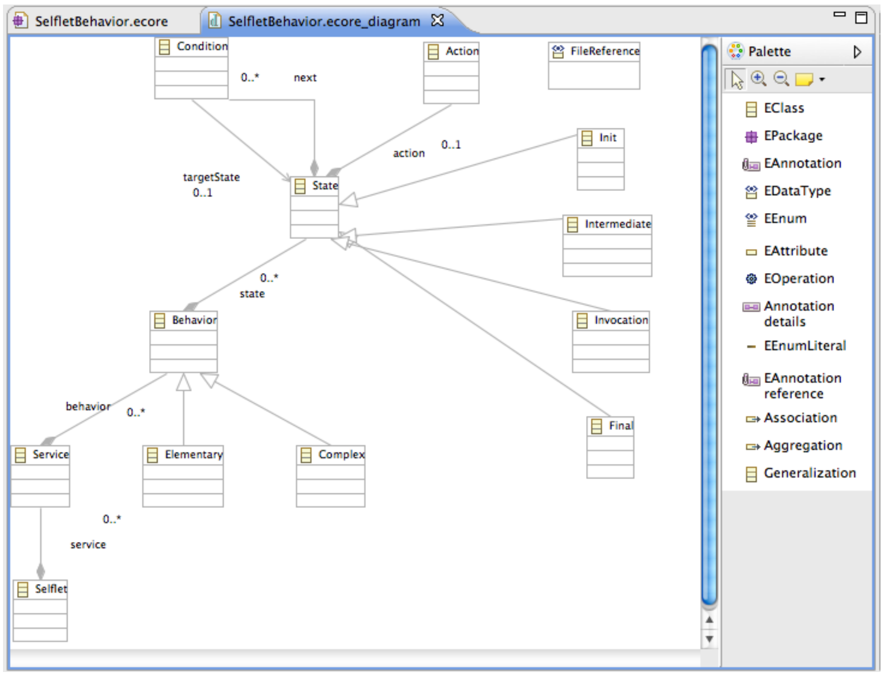
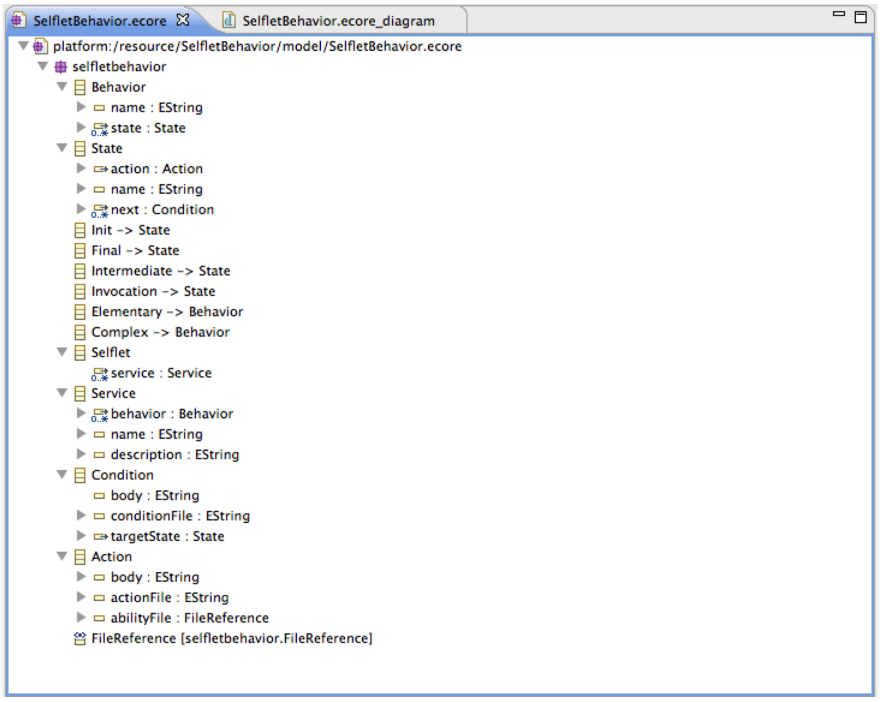


Figure 4.2: The Ecore Model diagram of our editor `SelfLetClipse2`



```
platform:/resource/SelfletBehavior/model/SelfletBehavior.ecore
  selfletbehavior
    Behavior
      name : EString
      state : State
    State
      action : Action
      name : EString
      next : Condition
    Init -> State
    Final -> State
    Intermediate -> State
    Invocation -> State
    Elementary -> Behavior
    Complex -> Behavior
    Selflet
      service : Service
    Service
      behavior : Behavior
      name : EString
      description : EString
    Condition
      body : EString
      conditionFile : EString
      targetState : State
    Action
      body : EString
      actionFile : EString
      abilityFile : FileReference
    FileReference [selfletbehavior.FileReference]
```

Figure 4.3: The Ecore Model of our editor `SelfLetClipse2`

Creation Tool and this Service domain model. In the next Chapter, we will see into details how the mapping is done.

One Service might contain many Behavior implementation, we first draw an abstract Behavior EClass, two concrete EClass Elementary and Complex that have the Generalization into the abstract Behavior class. Service has an Aggregation connection to Behavior so that Behavior and its concrete implementations can be put inside the Service rectangle.

Elementary Behavior has only 3 States: Initial State, Invocation State and Final State. Complex Behavior has 3 or more States: Initial State, one or many Intermediate State, and one or many Final State. This leads to the creation of an abstract State EClass and four concrete Init, Intermediate, Invocation, Final EClasses that have the Generalization connection into the State. Behavior EClasses have its Aggregation connection into State Class. In the next Chapter, we will see how we have mapped the Initial, Invocation and Final State to only the Elementary Behavior and Initial, Intermediate and Final State to only the Complex Behavior. Audit controls are used to limit the exact one Invocation State in Elementary Behavior.

On transition between States, there might be a condition, however, we might need to specify some properties of this condition such as the Condition File and Condition Body for displaying purpose. A separate EClass is thus created for Condition with "next" as the Aggregation connection from State to Condition. We might want to know on that Condition, which is the targetState so we specify an inverse connection called targetState from Condition to State, notice that this is just an Association from Condition to State, not an Aggregation connection.

Each State has an Action file which the Execution Manager uses to execute the Action within the State, we need to put that Action inside the State for viewing so we create an Action EClass and have an Aggregation of only 1 action for Action EClass, this makes us easy to put the additional properties for Action into Action EClass.

At this point, we have finished defining the meta-model for our graphical editor using the elements of the Ecore kernel models. The resulting meta-model is shown in Figure 4.2.

4.3 The EMF Generator Model

After we have the Ecore model for our `SelfLet` meta language, we can create the new EMF Generator Model by initializing the wizard New EMF Generator Model, select the Ecore model that we have created and name the new Generator Model. The EMF Generator Model is used to generate the model code, the factory classes and implementation classes in Java code for the Ecore Model. It also generates the Edit code, Editor code and test code for the Ecore Model we are developing. Figure 4.4 shows the Generator Model for `SelfLetClipse2` The commands available are described as follows:

Generate Model Code : Generate the Model Code for the Ecore Model that are located in the current project in the workspace, classes include: the Model Classes, the Factory Class and Adaptor Class. Because EMF and GEF follow the Model View Controller paradigm, this Model Classes can be seen as the Model in MVC programming model.

Generate Edit Code : Generate the Edit Code for the Ecore Model that are located in a separate project with the name formatted as *modelProject.edit*. This project contains the Provider classes for the Model, that handles the getter/setter methods for the Model Classes.

Generate Editor Code : Generate the Editor Code for the Ecore Model that are located in a separate project with the name formatted as *modelProject.editor*. This project contains the classes and functions for the Editor of the Model, all of the necessary extension points and dependencies for adding a complete Editor to the Eclipse Plugin Runtime Platform. We will see this in section 4.4

Generate Test Code : Generate the Test Code for the Model Classes in a separate project with the name formatted as follows: *modelProject.tests*.

Generate All : Generate all the codes in all projects described above, this is often the convenient command that is most useful when there are small changes in the Ecore Model that need to be updated to all the codes. In this implementation, we have chosen to use this method for our Model code, Model Edit code and Model Editor code because we do not make any changes to these ones. The customization for the diagram graphical editor happens only in the *modelProject.diagram* project, which is in the next Chapter.

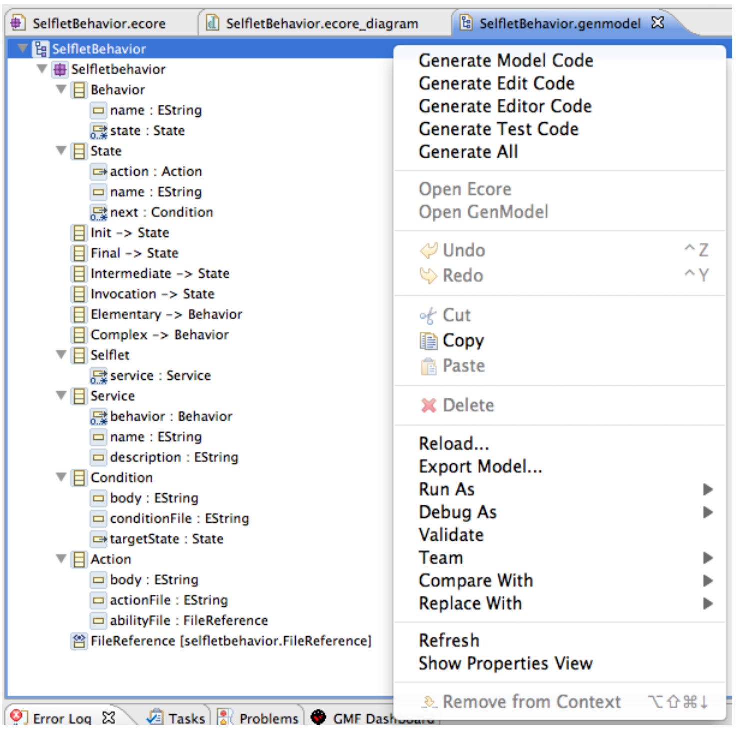
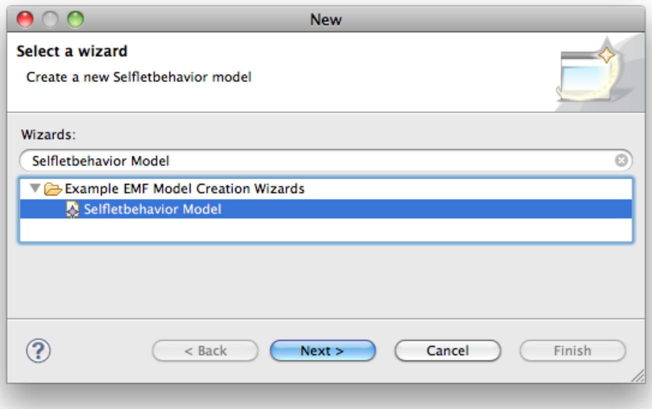


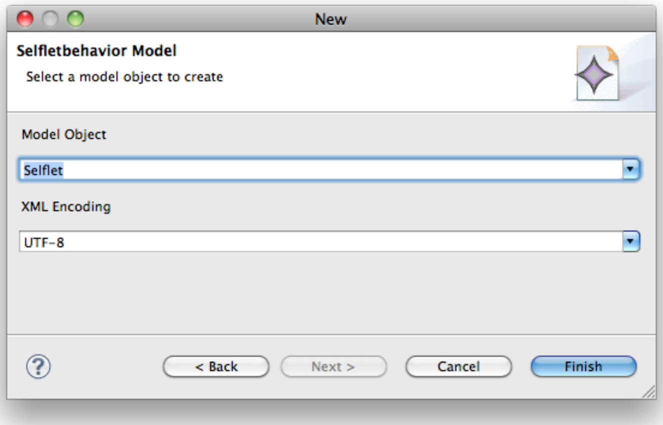
Figure 4.4: The `SelfLetClipse2` EMF Generator Model with the pop up menu

4.4 The EMF Editor

Because the EMF Framework is based on the general Eclipse Platform Framework so the Editor that it creates also enjoys all of the functionalities of the standard Eclipse Platform Environment, such as the Menus, Toolbars, Views, Perspectives, Project Explorer, etc... The Editor also has a wizard that allows developers to quickly add a new model file into the working project workspace. The Editor created allows us to right click on the Top Level Domain Element and add their children using the intuitive menu as in Figure 4.5. Developer can also view the created domain data model in XML Editor or in a Standard Eclipse Text Editor. The Figure 4.6 shows an example of the Editor in action, the persisted data is saved into the parallel XML file in the File System.



(a) The new SelfLetClipse2 EMF Model Wizard added to the standard Eclipse platform



Model Object: Selflet

XML Encoding: UTF-8

(b) The new SelfLetClipse2 EMF Model Wizard with selection of top level element

Figure 4.5: The new SelfLetClipse2 EMF Model Wizard

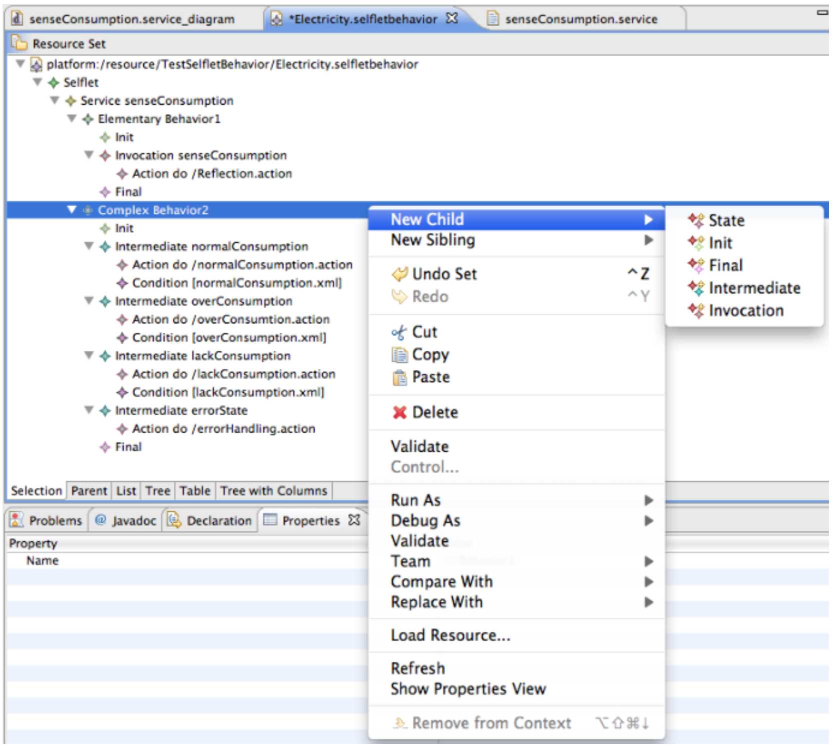

senseConsumption.service_diagram
*Electricity.selfletbehavior
senseConsumption.service
Resource Set
platform:/resource/TestSelfletBehavior/Electricity.selfletbehavior
Selflet
Service senseConsumption
Elementary Behavior1
Init
Invocation senseConsumption
Action do /Reflection.action
Final
Complex Behavior2
Init
Intermediate normalConsumption
Action do /normalConsumption.action
Condition [normalConsumption.xml]
Intermediate overConsumption
Action do /overConsumtion.action
Condition [overConsumption.xml]
Intermediate lackConsumption
Action do /lackConsumption.action
Condition [lackConsumption.xml]
Intermediate errorState
Action do /errorHandling.action
Final
Selection
Parent
List
Tree
Table
Tree with Columns
Problems
Javadoc
Declaration
Properties
Property
Name
New Child
New Sibling
Undo Set ^Z
Redo ^Y
Cut
Copy
Paste
Delete
Validate
Control...
Run As
Debug As
Validate
Team
Compare With
Replace With
Load Resource...
Refresh
Show Properties View
Remove from Context ⌥⇧⌘↓
State
Init
Final
Intermediate
Invocation

Figure 4.6: The `SelfLetClipse2` EMF Model Editor

4.5 Conclusion

In this Chapter, we have shown the meta-model that we are going to use to build the graphical editor for the `SelfLet` framework. Creating the EMF Editor correctly is the critical step towards the creation of the graphical editor using the GEF and GMF frameworks, because if the EMF Editor does not reflect well the domain data model then the next step would totally be a failure. In this Chapter, we have built the EMF Editor carefully and tested the result in the Eclipse Runtime Configuration Environment. In the next Chapter, we will see how the three frameworks EMF, GEF, GMF are employed to build the graphical editor for the `SelfLet` environment.

Chapter 5

The implementation details

5.1 Introduction

In this Chapter, we go through the steps for building the graphical editor using the combination of the three frameworks: EMF, GEF and GMF. We have seen in the previous Chapter how we used the EMF for creating the Ecore Model and the corresponding EMF Editor for our `SelfLetClipse2` graphical editor. In this Chapter, we go into details the remaining steps which include defining the graph, creating the tool, defining the mapping, transforming the mapping to the generator model, generating the diagram code, customizing the diagram code, adding Auditing containers and finally testing the diagram in the Eclipse Runtime Workspace. But first, in the next section, we talk about the installation prerequisites.

5.2 Installation prerequisites

We use Eclipse Helios the classic 3.6.2 version that can be downloaded from this link: [10], the reason for choosing this Eclipse Helios is the stability of the GMF Framework which enables us to focus our attention to developing the graphical editor but not on catching up with the newest capabilities that the frameworks provide. The result graphical editor is of course deployable into Eclipse Indigo, the latest release [11] at the time of writing. The second installation requirement is the Eclipse Modeling Tools which include the EMF, the GEF and the GMF. These frameworks can be installed using the Eclipse

Helios Update site from the standard Eclipse framework choosing the "Modeling" category. Figure 5.1 shows a screenshot of this dialog. The following lists the packages that need to be installed:

EMF - Eclipse Modeling Framework SDK : The full EMF SDK for development.

Graphical Editing Framework GEF SDK : The full GEF SDK for development.

Graphical Modeling Framework (GMF) Runtime SDK : The runtime for the GMF

Graphical Modeling Framework SDK : The full GMF SDK for development.

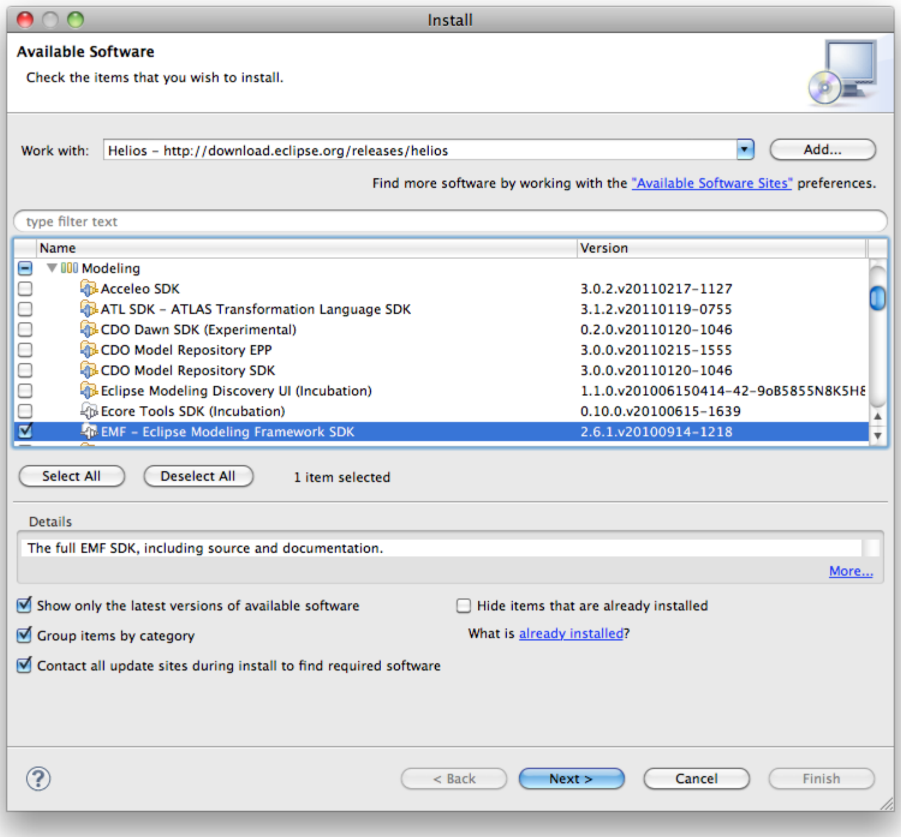
Besides that, we also need to install the "Plugins Development Tools" for various functionalities, project templates and perspectives that are suitable for developing an Eclipse Plugin.

That concludes the installation requirements for our graphical editor. The next Section goes into details the steps in creating a Graphical Editor using the GMF Framework.

5.3 Defining the Graphical Definition Model

In the previous Chapter, which we have shown the GMF Dashboard in Figure 4.1, we have examined the Domain Ecore Model (.ecore) and the Domain Gen Model (.genmodel) along with the EMF Editor. As we can see from the dashboard, the next step would be to define the graph or the graphical definition model (.gmfgraph).

The easiest way to create the Graphical definition Model is to let the GMF Framework to create automatically for us, however, we still have to add and customize a lot of items within the model created. Figure 5.2 shows the final result of our `SelfLetClipse2` editor's graphical definition model. To start the automatic process of creating the graphical definition model, the GMF Dashboard has a convenient command "Derive" from the "Domain Model" (.ecore) to launch the "Graphical Definition Model" wizard. Figure 5.3 shows



Helios - http://download.eclipse.org/releases/helios

Find more software by working with the "Available Software Sites" preferences.

	Name	Version
▣	Modeling	
☐	Acceleo SDK	3.0.2.v20110217-1127
☐	ATL SDK - ATLAS Transformation Language SDK	3.1.2.v20110119-0755
☐	CDO Dawn SDK (Experimental)	0.2.0.v20110120-1046
☐	CDO Model Repository EPP	3.0.0.v20110215-1555
☐	CDO Model Repository SDK	3.0.0.v20110120-1046
☐	Eclipse Modeling Discovery UI (Incubation)	1.1.0.v201006150414-42-9oB5855N8K5HE
☐	Ecore Tools SDK (Incubation)	0.10.0.v20100615-1639
☑	EMF - Eclipse Modeling Framework SDK	2.6.1.v20100914-1218

1 item selected

The full EMF SDK, including source and documentation.

☑ Show only the latest versions of available software

☑ Group items by category

☑ Contact all update sites during install to find required software

☐ Hide items that are already installed

What is already installed?

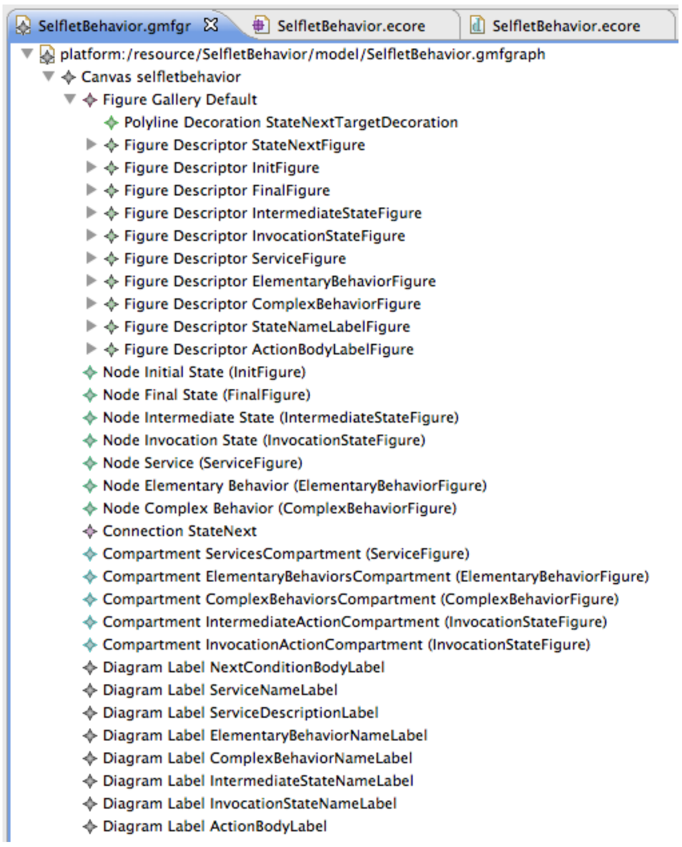
Figure 5.1: Eclipse Helios standard dialog for installing new software, the installation requirements are: EMF, GEF, GMF and the Plugins Development Tools

an example of definition wizard when we have selected the top level domain element as the `SelfLet`. The following list shows the definition of the available selection in the wizard, as well as the graphical definition created.

Node : The selected domain element is created as a Node, the standard creation is a FigureDescriptor with a rectangle and a label. The rectangle represents the graphical definition for the domain element selected, while the label might be one of its attributes.

Link : The selected domain element is created as a Link, usually it is a polyline connection without the decorator

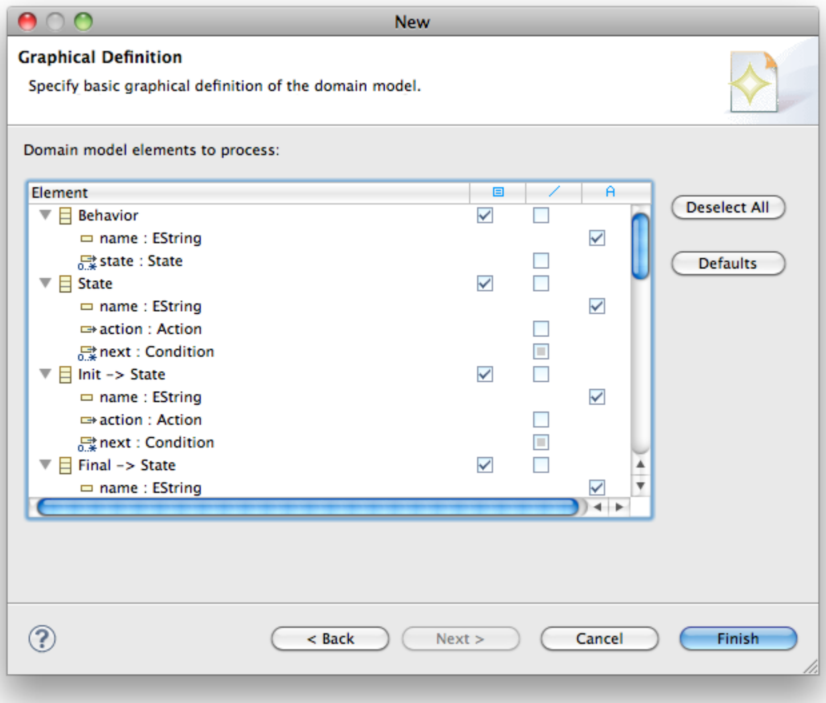
Label : The selected attribute is created as a Label in the graphical definition.



```
SelfletBehavior.gmfgr    SelfletBehavior.ecore    SelfletBehavior.ecore
platform:/resource/SelfletBehavior/model/SelfletBehavior.gmfgraph
  Canvas selfletbehavior
    Figure Gallery Default
      Polyline Decoration StateNextTargetDecoration
      Figure Descriptor StateNextFigure
      Figure Descriptor InitFigure
      Figure Descriptor FinalFigure
      Figure Descriptor IntermediateStateFigure
      Figure Descriptor InvocationStateFigure
      Figure Descriptor ServiceFigure
      Figure Descriptor ElementaryBehaviorFigure
      Figure Descriptor ComplexBehaviorFigure
      Figure Descriptor StateNameLabelFigure
      Figure Descriptor ActionBodyLabelFigure
    Node Initial State (InitFigure)
    Node Final State (FinalFigure)
    Node Intermediate State (IntermediateStateFigure)
    Node Invocation State (InvocationStateFigure)
    Node Service (ServiceFigure)
    Node Elementary Behavior (ElementaryBehaviorFigure)
    Node Complex Behavior (ComplexBehaviorFigure)
    Connection StateNext
    Compartment ServicesCompartment (ServiceFigure)
    Compartment ElementaryBehaviorsCompartment (ElementaryBehaviorFigure)
    Compartment ComplexBehaviorsCompartment (ComplexBehaviorFigure)
    Compartment IntermediateActionCompartment (InvocationStateFigure)
    Compartment InvocationActionCompartment (InvocationStateFigure)
    Diagram Label NextConditionBodyLabel
    Diagram Label ServiceNameLabel
    Diagram Label ServiceDescriptionLabel
    Diagram Label ElementaryBehaviorNameLabel
    Diagram Label ComplexBehaviorNameLabel
    Diagram Label IntermediateStateNameLabel
    Diagram Label InvocationStateNameLabel
    Diagram Label ActionBodyLabel
```

Figure 5.2: The `SelfLetClipse2` Graphical Definition Model

After the generation phase, we might have the graphical definition that is not really suitable for our purpose but with basic structure. The top level element is the Canvas that the user is going to draw on, on this canvas, there are



New

Graphical Definition

Specify basic graphical definition of the domain model.

Domain model elements to process:

Element			
▼ Behavior	✓	☐	
name : EString			✓
state : State		☐	
▼ State	✓	☐	
name : EString			✓
action : Action		☐	
next : Condition		▣	
▼ Init -> State	✓	☐	
name : EString			✓
action : Action		☐	
next : Condition		▣	
▼ Final -> State	✓	☐	
name : EString			✓

Figure 5.3: The Graphical Definition Model wizard assistance

several children, the most interesting and difficult to construct is the Figure Gallery. This is where all the definitions of the graphical editor is created. The following shows a list of items we have used in the graphical definition for `SelfLetClipse2`. Of course there might be rooms for improvement in this graphical definition for the editor to look better.

Figure Gallery : contains all the Figure Descriptor for the graphical editors, this is the first place that we have to make modifications and additions. In the `SelfLetClipse2`, we have the Figure Descriptors for the following items:

ServiceFigure : This is the top level Figure that is a rectangle and contains a label for Service Name and a compartment for its corresponding Elementary Behavior Figure and Complex Behavior Figure. Elementary Behavior Figure and Complex Behavior Figure are positioned vertically by customizing the diagram code later in the next Chapter.

- **ElementaryBehaviorFigure** : This is the Figure for the Elementary Behavior, it is a rounded rectangle with a label for Behavior name and a rectangle compartment for storing inside its States.
- **ComplexBehaviorFigure** : This is the Figure for the Complex Behavior, it is also a rounded rectangle with a label for Behavior name and a rectangle compartment for storing its various States.
- **InitFigure** : The Figure for the Initial State in the Elementary Behavior Figure or Complex Behavior Figure, it is an Ellipse with black background with small size.
- **FinalFigure** : The Figure for the Final State in the Behavior Figure, it is an Ellipse with an ellipse inside and bigger size than the Initial State.
- **IntermediateStateFigure** : The Intermediate State in the Complex Behavior Figure, it is a rectangle with no label but contains another rectangle compartment for storing Action Figure.
- **InvocationStateFigure** : The Invocation State in the Elementary Behavior Figure, it is a rectangle with no label but contains another rectangle compartment for storing the Action Figure.
- **StateNameLabelFigure** : The Label for the State Name
- **ActionBodyLabelFigure** : The Label for the Action Body
- **StateNextFigure** : The polyline connection Figure for transition between States in the Behavior Figure. It uses a decoration on one side of the connection for enabling the arrow like connection.

Node : The Actual Node drawable in the Canvas, it specifies the Figure Descriptor as described earlier. So in our Canvas, there are the following Nodes: Initial State Node, Final State Node, Intermediate State Node, Invocation State Node, Service Node, Elementary Behavior Node, Complex Behavior Node.

Connection : The Actual Connection drawable in the Canvas, in our Canvas, there is only one Connection that connect between States in the Behavior Figure.

Compartment : The Actual Compartment drawable in the Canvas, we have in total 5 compartments: Service Compartment, Elementary Behavior Compartment, Complex Behavior Compartment, Intermediate State Compartment, Invocation State Compartment.

Diagram Label : The actual Diagram Label that can be drawn in the Canvas.

With the Graphical Definition Model ready, we are ready to create the Creation Tool for the Palette in the next Section.

5.4 Creating the Palette Creation Tool

The Palette Creation Tools are the tools that the user can use to draw an item into the Canvas. The GMF Dashboard can be used to assist the creation of the Creation Tools. The "Derive" command from the Domain Model to Creation Tool in the GMF Dashboard enables us to create automatically our desired Creation Tools.

In `SelfLetClipse2`, we envisioned a diagram palette with the following creation tools: (The order has been changed to reflect the logic of the `SelfLetClipse2` editor)

Elementary Behavior : Creation Tool for drawing the Elementary Behavior inside the Service

Complex Behavior : Creation Tool for drawing the Complex Behavior inside the Service

Initial State : Creation Tool for drawing the Initial State inside the Elementary Behavior or Complex Behavior

Final State : Creation Tool for drawing the Final State inside the Elementary Behavior or Complex Behavior.

Intermediate State : Creation Tool for drawing the Intermediate State inside the Complex Behavior.

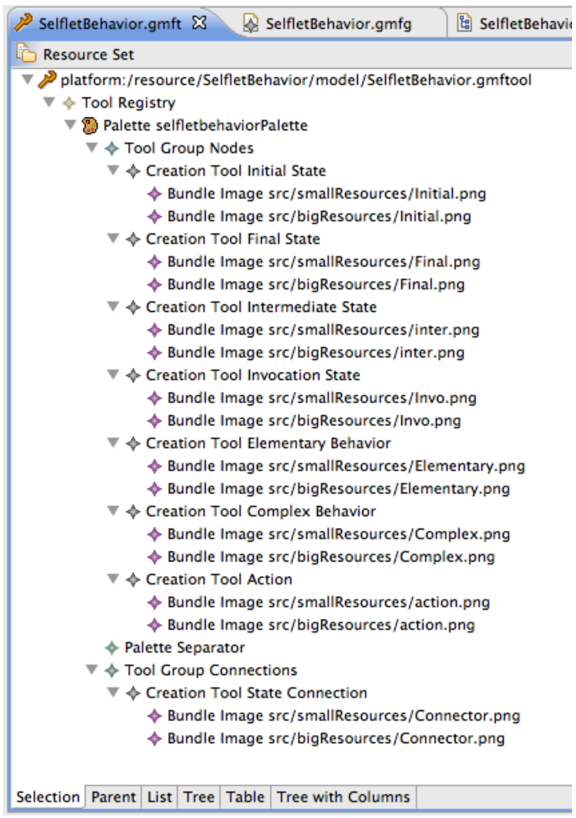
Invocation State : Creation Tool for drawing the Invocation State inside the Elementary Behavior.

Action : Creation Tool for drawing the Action inside the Elementary Behavior or Complex Behavior.

State Connection : Creation Tool for drawing the Connection between the States.

There is no Creation Tool for Service, as we create one Service by default when user creates a new graph. We will see how to do that in Section 5.8.

With each Creation Tool, we can specify the Default Image or Bundle Image for the small icon and big icon. These icons files are imported into the project sources as resources files. The Figure 5.4 shows the final result of our Creation Tool.



```
Resource Set
▼ platform:/resource/SelfletBehavior/model/SelfletBehavior.gmftool
  ▼ Tool Registry
    ▼ Palette selfletbehaviorPalette
      ▼ Tool Group Nodes
        ▼ Creation Tool Initial State
            Bundle Image src/smallResources/Initial.png
            Bundle Image src/bigResources/Initial.png
        ▼ Creation Tool Final State
            Bundle Image src/smallResources/Final.png
            Bundle Image src/bigResources/Final.png
        ▼ Creation Tool Intermediate State
            Bundle Image src/smallResources/inter.png
            Bundle Image src/bigResources/inter.png
        ▼ Creation Tool Invocation State
            Bundle Image src/smallResources/Invo.png
            Bundle Image src/bigResources/Invo.png
        ▼ Creation Tool Elementary Behavior
            Bundle Image src/smallResources/Elementary.png
            Bundle Image src/bigResources/Elementary.png
        ▼ Creation Tool Complex Behavior
            Bundle Image src/smallResources/Complex.png
            Bundle Image src/bigResources/Complex.png
        ▼ Creation Tool Action
            Bundle Image src/smallResources/action.png
            Bundle Image src/bigResources/action.png
      Palette Separator
      ▼ Tool Group Connections
        ▼ Creation Tool State Connection
            Bundle Image src/smallResources/Connector.png
            Bundle Image src/bigResources/Connector.png
```

Figure 5.4: The `SelfLetClipse2` Palette Creation Tool Definition

5.5 Determining the Mapping

The next step as shown in the GMF Dashboard is combining the Domain Model, the Graphical Definition Model and the Creation Tools into the Mapping Model (.gmfmap). This mapping model is the way we tell the GMF

Framework how we would like to combine our domain model with the graphical model and the creation tool. The "Combine" command from the three elements into the Mapping Model in the GMF Dashboard helps us to initialize the wizard to do the mapping. The final result of the `SelfLetClipse2` Mapping Model is shown in Figure 5.5. We briefly describe the Mapping for our `SelfLetClipse2` Editor as it is the most important part of designing the graphical editor using the GMF Framework.

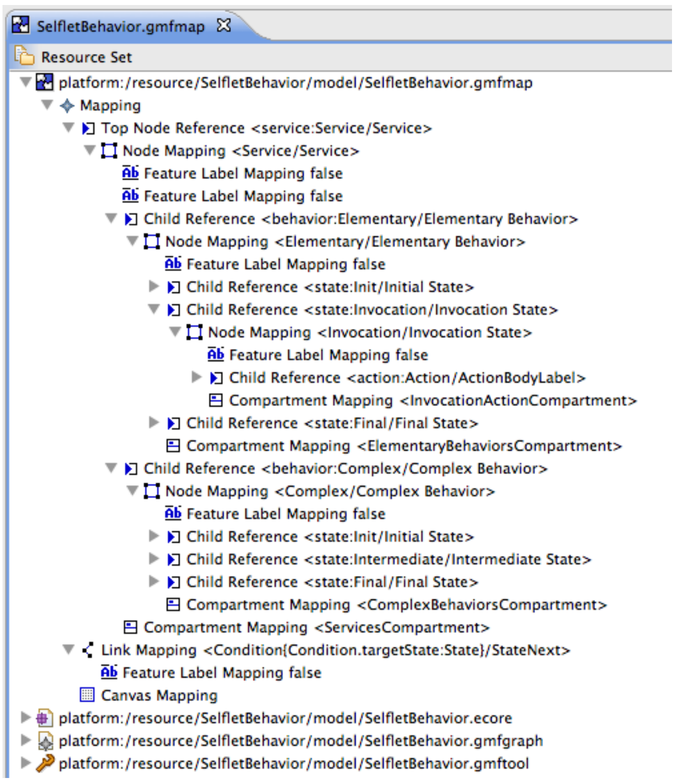
Mapping : The container for the Mapping of elements, it is obligatory element.

Canvas Mapping : This is the first element in the mapping, it specifies which domain model and element we are working on, in our case, it is the SelfLet. It specifies which Palette we are working on, that is the Creation Palette we have created earlier. It also specifies which Diagram Canvas we are drawing into, that is the graphical definition model we have created earlier.

Top Node Reference : The outer most mapping element in the editor, in this mapping, we have chosen to specify the Containment Feature as the Selflet.service:Service so that the first node to draw in the diagram is the Service. A little explanation is necessary here about the Containment Feature, it is known as by-value aggregation in UML, containment is a stronger type of association that implies a whole-part relationship: an object cannot, directly or indirectly, contain its own container; it can have no more than one container and its life span ends with that of its container.

Node Mapping : The first Node Mapping for Service contains the labels for Service Name and two Child References for Elementary Behavior and Complex Behavior. Elementary Behavior in turn contains the Node Mapping with a label and three Child Reference for Initial State, the Invocation State and the Final State. Complex Behavior contains the Node Mapping with a label and three Child Reference for Initial State, the Intermediate State and the Final State. Child Reference for Invocation State and Intermediate State are worth talking about because they contain Node Mapping with Child Reference to Action. Each Action then has its own Node Mapping to the Diagram Label ActionBodyLabel. Each Node Mapping has its own Element in the Domain Model, its Diagram Node in the Graphical Definition Model and its Creation in the Palette. The concept continues with other Node Mapping and Child Reference.

Link Mapping : Link Mapping specifies what happen when user uses the Palette Creation Tool to connect between one Node in the Diagram to the other. In our `SelfLetClipse2`, we use only the connections between States, so, the containment feature is the State.next:Condition, the Element is the Condition Element, the Target Feature is the Condition.targetState:State, the Creation Tool is the State Connection in the Palette and the Diagram Link is the Connection StateNext.



```
SelfletBehavior.gmfmap
Resource Set
▼ platform:/resource/SelfletBehavior/model/SelfletBehavior.gmfmap
  ▼ Mapping
    ▼ Top Node Reference <service:Service/Service>
      ▼ Node Mapping <Service/Service>
          Feature Label Mapping false
          Feature Label Mapping false
        ▼ Child Reference <behavior:Elementary/Elementary Behavior>
          ▼ Node Mapping <Elementary/Elementary Behavior>
              Feature Label Mapping false
            ▶ Child Reference <state:Init/Initial State>
            ▼ Child Reference <state:Invocation/Invocation State>
              ▼ Node Mapping <Invocation/Invocation State>
                  Feature Label Mapping false
                ▶ Child Reference <action:Action/ActionBodyLabel>
                  Compartment Mapping <InvocationActionCompartment>
            ▶ Child Reference <state:Final/Final State>
              Compartment Mapping <ElementaryBehaviorsCompartment>
        ▼ Child Reference <behavior:Complex/Complex Behavior>
          ▼ Node Mapping <Complex/Complex Behavior>
              Feature Label Mapping false
            ▶ Child Reference <state:Init/Initial State>
            ▶ Child Reference <state:Intermediate/Intermediate State>
            ▶ Child Reference <state:Final/Final State>
              Compartment Mapping <ComplexBehaviorsCompartment>
          Compartment Mapping <ServicesCompartment>
    ▼ Link Mapping <Condition{Condition.targetState:State}/StateNext>
        Feature Label Mapping false
      Canvas Mapping
▶ platform:/resource/SelfletBehavior/model/SelfletBehavior.ecore
▶ platform:/resource/SelfletBehavior/model/SelfletBehavior.gmfgraph
▶ platform:/resource/SelfletBehavior/model/SelfletBehavior.gmftool
```

Figure 5.5: The `SelfLetClipse2` Mapping Model

In this mapping, we can also define the Audits Container which is used to validate the graphical diagram that users create with our specific constraints, for example, to restrict only one Invocation State can be inserted into the

Elementary Behavior. The language that is used to define the Audits Control is OCL - The Object Constraint Language [17]

5.6 Transforming the Generator Model

After we have finished defining the Mapping Model for our `SelfLetClipse2` editor, we can use the convenient command in the GMF Dashboard called "Transform" to create the Diagram Editor Generator Model (.gmfgen). This is the helper file in creating the diagram code project. In this Generator Model, we can specify the domain model file extension, the graphical editor file extension, apart from many other features of the graphical editor that we are going to create. While transforming the mapping model to the generator model, GMF also allows us to transform into a Rich Client Platform RCP[14]. In this implementation, we have chosen to implement `SelfLetClipse2` as a Plugin for the Eclipse Platform, so we do not choose RCP as the option.

5.7 Creating the Diagram project

The Diagram project can be created using the command "Generate Diagram Editor" available in the GMF Dashboard. This activates the process of generating the source code for the diagram project in the workspace, the name format is "modelProject.diagram". Up to now we have five projects in the workspace "modelProject", "modelProject.edit", "modelProject.editor", "modelProject.tests", "modelProject.diagram". If there are no build errors in the workspace (normally the generated code does not contain any errors if we do everything correctly, otherwise it informs us where the errors are). We can start a new Runtime Configuration to test the editor and add some test diagrams by using the wizard created by the diagram project. However, we are not able to add anything into the Drawing Canvas, as we have chosen to not include the Service Creation Tool. Next step, we have to customize the Diagram code to do our modification.

5.8 Customizing the Diagram

The following customizations have been made to the Diagram project in order to get the desired Editor as we have seen in Chapter 3.

- Adding automatically the Service Figure when the new Diagram is created.
- Creating the Vertical Layout for the Elementary Behavior and Complex Behavior in the Service Figure.
- Enabling the File Selector in the Properties view in order for selection of the Abilities File with .jar extension
- Copying automatically the Abilities .jar file when user selects it from the dialog and putting it into the project workspace and storage file system.
- Creating automatically the Action file template and Action Body when an Action is added in the State in the Elementary Behavior or Complex Behavior.

The source code shown the specific actions taken for these customization are omitted because later in Section 5.10, we will talk about the source version control used in `SelfLetClipse2` that enables us to see the differences we made to each commits.

5.9 Integrating with the previous `SelfLetClipse`

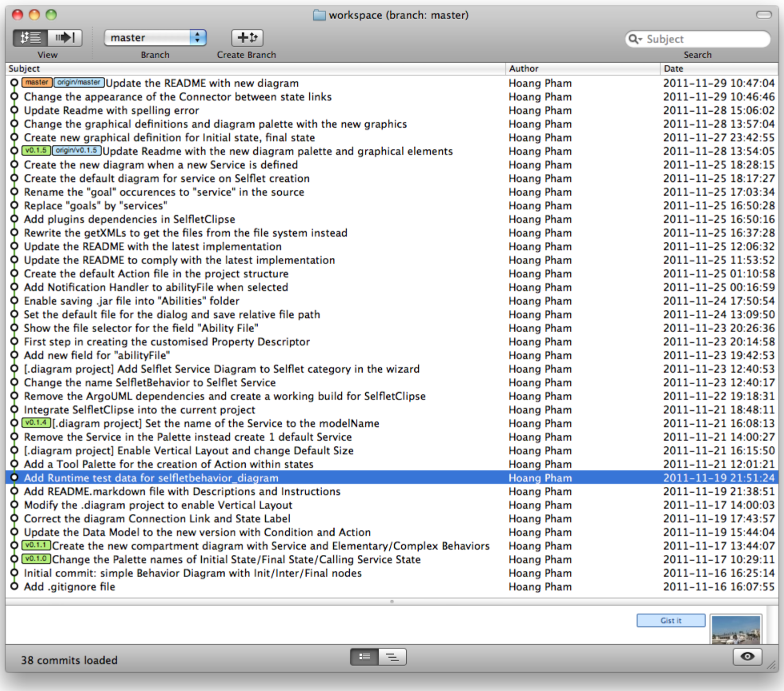
The IDE `SelfLetClipse` developed by Calcavecchia [4, 2, 5] has integrated three wizards for the creation of `SelfLet` project, `SelfLet` Goal and `SelfLet` Behavior. In this new editor `SelfLetClipse2`, we would like to integrate one existing wizard: the `SelfLet` wizard to create the `SelfLet` project in the workspace, we take dependencies of `SelfLetClipse2` for the previous `SelfLetClipse` and rewrite the "performFinish" function of the wizard to create the new project structure of the `SelfLet` and new diagram with the new editor.

We rename the `SelfLet` Goal with the `SelfLet` Service in order to let `SelfLet` developers to add new Service into the project, however, at the end of the wizard, we rewrite the "performFinish" function to create the new diagram and diagram domain model based on the new editor implemented.

In doing this, the existing `SelfLet` developers have little difficulties in getting familiar with the new editor and project structure.

5.10 The source version control

We have implemented `SelfLetClipse2` in Eclipse and we used heavily Git[6] as our source control system, each small change is committed to Git so that the committed message makes sense. The project is uploaded into GitHub[16] as the repository for sharing and collaborating. Up to the time of writing, 38 commits have been created for `SelfLetClipse2` as can be seen in Figure 5.6. Because the GMF Framework uses the generated code as the main methods for creating diagram editors, some of our modifications might be lost if we tried to delete the whole project and regenerate again, one can use the Rewrite History function of Git to move the commits with [.diagram project] message to the top. By doing this, the changes made specifically to diagram project code remain.



Subject	Author	Date
master origin/master Update the README with new diagram	Hoang Pham	2011-11-29 10:47:04
Change the appearance of the Connector between state links	Hoang Pham	2011-11-29 10:46:46
Update Readme with spelling error	Hoang Pham	2011-11-28 15:06:02
Change the graphical definitions and diagram palette with the new graphics	Hoang Pham	2011-11-28 13:57:04
Create new graphical definition for Initial state, final state	Hoang Pham	2011-11-27 23:42:55
v0.1.5 origin/v0.1.5 Update Readme with the new diagram palette and graphical elements	Hoang Pham	2011-11-28 13:54:05
Create the new diagram when a new Service is defined	Hoang Pham	2011-11-25 18:28:15
Create the default diagram for service on Selflet creation	Hoang Pham	2011-11-25 18:17:27
Rename the "goal" occurences to "service" in the source	Hoang Pham	2011-11-25 17:03:34
Replace "goals" by "services"	Hoang Pham	2011-11-25 16:50:28
Add plugins dependencies in SelfletClipse	Hoang Pham	2011-11-25 16:50:16
Rewrite the getXMLs to get the files from the file system instead	Hoang Pham	2011-11-25 16:37:28
Update the README with the latest implementation	Hoang Pham	2011-11-25 12:06:32
Update the README to comply with the latest implementation	Hoang Pham	2011-11-25 11:53:52
Create the default Action file in the project structure	Hoang Pham	2011-11-25 01:10:58
Add Notification Handler to abilityFile when selected	Hoang Pham	2011-11-25 00:16:59
Enable saving .jar file into "Abilities" folder	Hoang Pham	2011-11-24 17:50:54
Set the default file for the dialog and save relative file path	Hoang Pham	2011-11-24 13:09:50
Show the file selector for the field "Ability File"	Hoang Pham	2011-11-23 20:26:36
First step in creating the customised Property Descriptor	Hoang Pham	2011-11-23 20:14:58
Add new field for "abilityFile"	Hoang Pham	2011-11-23 19:42:53
[.diagram project] Add Selflet Service Diagram to Selflet category in the wizard	Hoang Pham	2011-11-23 12:40:53
Change the name SelfletBehavior to Selflet Service	Hoang Pham	2011-11-23 12:40:17
Remove the ArgoUML dependencies and create a working build for SelfletClipse	Hoang Pham	2011-11-22 19:18:31
Integrate SelfletClipse into the current project	Hoang Pham	2011-11-21 18:48:11
v0.1.4 [.diagram project] Set the name of the Service to the modelName	Hoang Pham	2011-11-21 16:08:13
Remove the Service in the Palette instead create 1 default Service	Hoang Pham	2011-11-21 14:00:27
[.diagram project] Enable Vertical Layout and change Default Size	Hoang Pham	2011-11-21 16:15:50
Add a Tool Palette for the creation of Action within states	Hoang Pham	2011-11-21 12:01:21
Add Runtime test data for selfletbehavior_diagram	Hoang Pham	2011-11-19 21:51:24
Add README.markdown file with Descriptions and Instructions	Hoang Pham	2011-11-19 21:38:51
Modify the .diagram project to enable Vertical Layout	Hoang Pham	2011-11-17 14:00:03
Correct the diagram Connection Link and State Label	Hoang Pham	2011-11-19 17:43:57
Update the Data Model to the new version with Condition and Action	Hoang Pham	2011-11-19 15:44:04
v0.1.1 Create the new compartment diagram with Service and Elementary/Complex Behaviors	Hoang Pham	2011-11-17 13:44:07
v0.1.0 Change the Palette names of Initial State/Final State/Calling Service State	Hoang Pham	2011-11-17 10:29:11
Initial commit: simple Behavior Diagram with Init/Inter/Final nodes	Hoang Pham	2011-11-16 16:25:14
Add .gitignore file	Hoang Pham	2011-11-16 16:07:55

38 commits loaded

Figure 5.6: The Git Commit History of the graphical editor `SelfLetClipse2`

The `SelfLetClipse2` project is located at: [20]

5.11 Conclusion

In this Chapter, we have gone through the whole process of creating the complete graphical editor using the GMF Dashboard as an assistance to our process, however, modifications are necessary and customizations to the final graphical editor also requires some investigations. We have strongly been convinced that by combining the three frameworks EMF, GEF and GMF, we can create astonishing graphical editors for any domain model and use the data underneath for various processing and management purpose.

In the next Chapter, we will walk through some examples of creating the graphical diagrams for Service with Elementary Behavior and Service with Complex Behavior.

Chapter 6

The example of creating a `SelfLet` project

6.1 Introduction

In order to show the advantages of using the newly developed integrated development environment, in this Chapter, we will show the main characteristics of `SelfLetClipse2` using an example of creating different `SelfLet` projects to be deployed in the Cloud. Three `SelfLets` to be created are: `SelfLet` with cloud optimization policy, Cloud Manager , Cloud `SelfLet`. In the paper by Calcavecchia et. al.[3] "Developing applications in the Cloud through the `SelfLet` framework", these entities have been explained very clearly, here, we summarize the description:

`SelfLet` with cloud optimization policy : the `SelfLet` that provides services whose execution is independent of the physical hosting context.

Cloud Manager : the most important `SelfLet` and owns the information required to access the cloud infrastructure. It interacts with the Infrastructure-as-a-Service provider API to start, terminate and manage the Virtual Machine instances execution.

Cloud `SelfLet` : the `SelfLet` that is hosted on a Virtual Machine instance in the cloud.

6.2 Project initialization

The SelfLet project is created using the SelfLet wizard that is available in SelfLet category when user selects the menu File -> New -> Other... of the standard Eclipse framework. Figure 6.1 shows the screenshot of the SelfLet category along with three items:

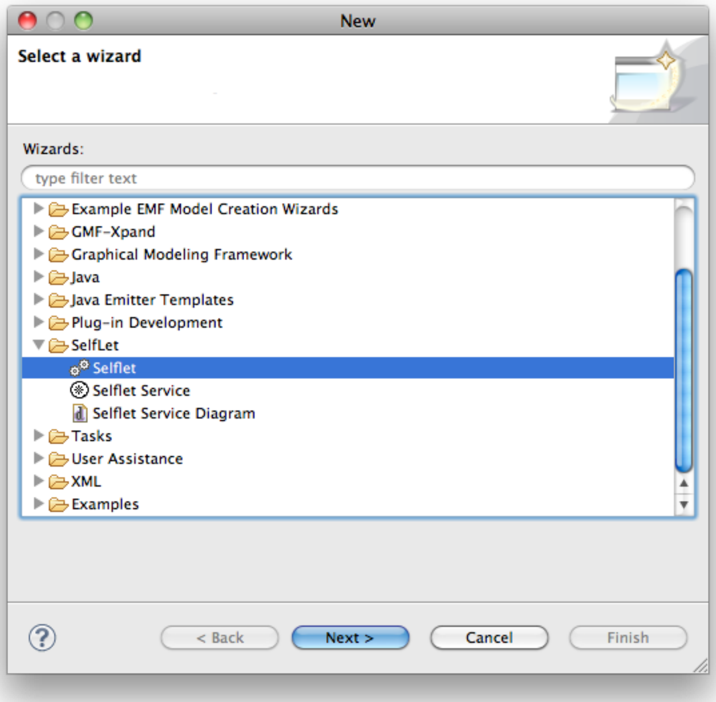


Figure 6.1: The SelfLetClipse2 SelfLet category with three items: SelfLet, SelfLet Service, SelfLet Service Diagram

SelfLet project is created using the SelfLet wizard. Here we describe each of the wizards available in the Figure:

SelfLet : For creating the SelfLet project in the workspace. This wizard guides the developers from defining the SelfLet properties to adding the main Service into the SelfLet. After finishing the wizard, a SelfLet project with the basic structure is created and a service diagram and service data model are created.

SelfLet **Service** : For creating new Service and Service Diagram at the same time.

`SelfLet` **Service Diagram** : For creating only Service Diagram, this Service Diagram may serve as the reference for other behavior implementation intermediate state in other `SelfLet`.

By selecting the `SelfLet` wizard, the next step in the wizard is defining the properties of the `SelfLet`.

Figure 6.2 shows how we define the properties of the different `SelfLet` projects.

After specifying the `SelfLet` properties, developers are asked to give the main Service description. This includes the Service name and its inputs, output parameters. Output parameter is specified using the checkbox before the parameter name (It's how the previous `SelfLetClipse` was done.)

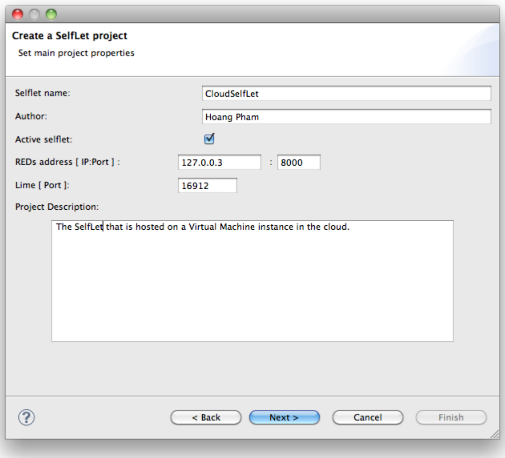
Figure 6.3 shows the description of the Video Search Service for the Cloud Optimization Policy `SelfLet` with its name and input, output parameters.

After finishing the `SelfLet` wizard, the standard `SelfLet` project is created in workspace with the following folders:

- *abilities*: the folder container for storing the abilities files in .jar format. These files are called by the Invocation States in the Elementary Behavior's implementation. When first created, this folder does not contain any item, `SelfLet` developers have to develop the specific task for the elementary behavior and use the File Selector in the Invocation State's Action property to copy the ability file into this folder. The files are then copied into the project workspace and in parallel into the file system.

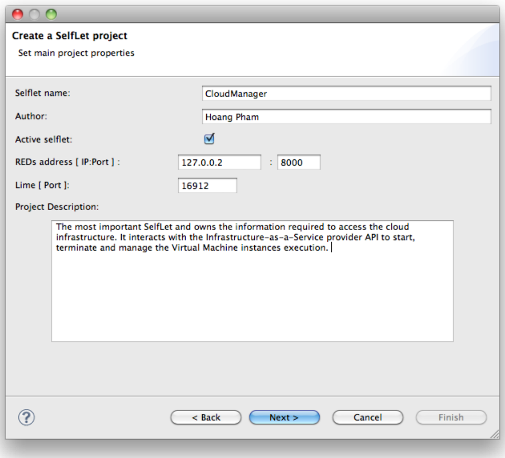
- *actions*: contains the actions of the States in the Behavior Implementations of all the Services inside the current `SelfLet`. This action files are block of codes that serves as the template for adding further Actions using the standard java language. Parameters in Action files are substituted by the Execution Manager each time the Execution Manager parses successfully the parameters in the Action files.

- *behaviors*: contains the diagram files and domain model files for the diagram editor of Behavior Implementation of the Services. Diagram file is edited using the "Service Diagram Editing" Editor that we created in the last Chapter, the elements in the Diagram are saved in parallel into the xml domain model file.


Create a SelfLet project
Set main project properties

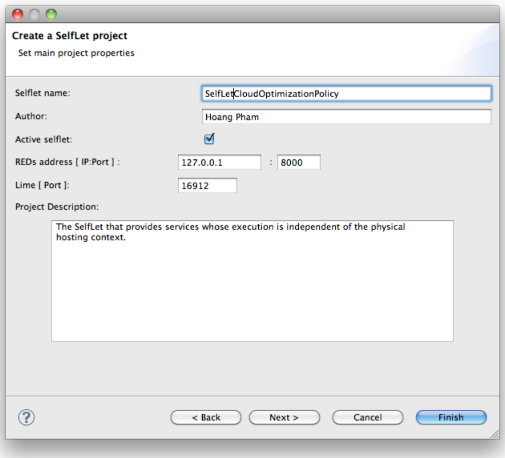
Selflet name:	CloudSelfLet
Author:	Hoang Pham
Active selflet:	✓
REDs address [IP:Port] :	127.0.0.3 : 8000
Lime [Port]:	16912
Project Description:	The SelfLet that is hosted on a Virtual Machine instance in the cloud.

(a) Cloud SelfLet Optimization Policy


Create a SelfLet project
Set main project properties

Selflet name:	CloudManager
Author:	Hoang Pham
Active selflet:	✓
REDs address [IP:Port] :	127.0.0.2 : 8000
Lime [Port]:	16912
Project Description:	The most important SelfLet and owns the information required to access the cloud infrastructure. It interacts with the Infrastructure-as-a-Service provider API to start, terminate and manage the Virtual Machine instances execution.

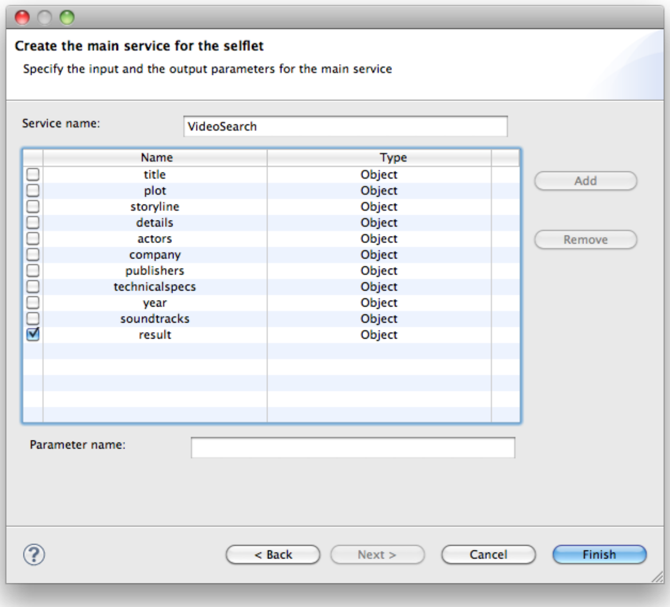
(b) Cloud Manager


Create a SelfLet project
Set main project properties

Selflet name:	SelfLetCloudOptimizationPolicy
Author:	Hoang Pham
Active selflet:	✓
REDs address [IP:Port] :	127.0.0.1 : 8000
Lime [Port]:	16912
Project Description:	The SelfLet that provides services whose execution is independent of the physical hosting context.

(c) SelfLet Cloud

Figure 6.2: The creation of three SelfLet projects using the wizard



Create the main service for the selflet

Specify the input and the output parameters for the main service

Service name: VideoSearch

	Name	Type
☐	title	Object
☐	plot	Object
☐	storyline	Object
☐	details	Object
☐	actors	Object
☐	company	Object
☐	publishers	Object
☐	technicalspecs	Object
☐	year	Object
☐	soundtracks	Object
✓	result	Object

Parameter name:

Figure 6.3: The `SelfLetClipse2` Service description

- *conditions*: contains the condition files that are written in XML that specifies the transitioning condition between one state and the other in the Elementary Behavior or Complex Behavior implementations of a Service.
- *rules*: contains the rules for the entire `SelfLet` project, rules are written in Drools language.
- *services*: contains the services description of the `SelfLet` project, services descriptions are stored as XML file with name, input, output properties.
- *selflet.xml*: contains the xml description of the `SelfLet` project, properties that developers declared in the wizard, all the files and folders existing in the `SelfLet` project.

Figure 6.4 shows the screenshot of the project after creation using the `SelfLet` wizard. It contains the folder structures as described above, and a default Service container for adding implementation behaviors into it. This default Service has the name the same as the name of the Service domain model file, which is: serviceName.service. In the next Section, we add the Complex Behaviors for the corresponding `SelfLet` projects.

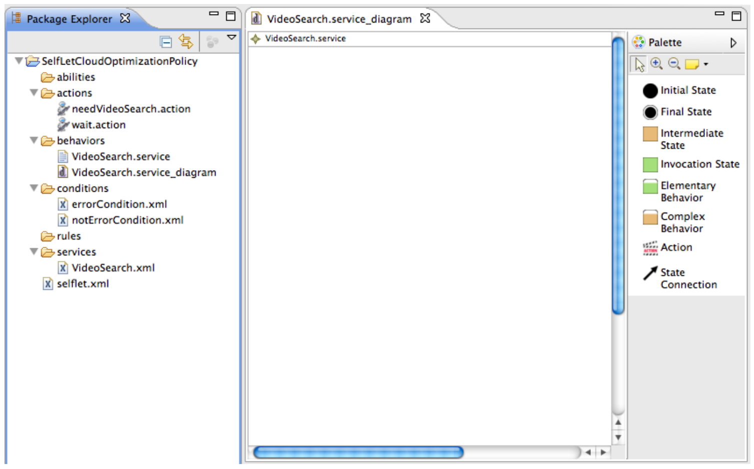


Figure 6.4: The `SelfLetClipse2` initial project creation template

6.3 Adding Complex Behavior Implementation

The creation tool "Complex Behavior" in the Palette is used to add Complex Behavior implementation into a Service. It is also positioned vertically inside the Service rectangle. Adding more Complex Behavior is done by selecting the Complex Behavior and clicking on the area outside of the existing Elementary Behaviors or Complex Behaviors of the Service.

Complex Behavior has its Name displayed as a label and the property Name can be changed immediately when the Complex Behavior is added into the diagram or with the properties View of the Eclipse Platform (Eclipse -> Window -> Show View -> Other -> Properties).

6.3.1 Adding States into the Complex Behavior Implementation

Complex Behavior implementations are also expressed using the State Diagrams the same as Elementary Behavior. The following list shows the steps which can be used to add States into the Complex Behavior:

- Use the creation tool "Initial State" to add the Initial State into the Complex Behavior implementation the same as Elementary Behavior.

- Use the creation tool "Intermediate State" to add the Intermediate State into the Complex Behavior implementation. Developer needs to specify its Name, so this Name can be used to create the Action file path and Action file template. Many Intermediate States can be added into the Complex Behavior diagram without causing the `SelfLetClipse2` IDE to validate any errors. These Intermediate States can have connection between each others.

- Use the creation tool "Action" to add the actual Action into this Intermediate State. Action for this Intermediate State contains only the Action file property and Action body.
- "Action file" property is created automatically by the `SelfLetClipse2` IDE. The format of the Action file template created in the project workspace is the same as in the Elementary Behavior section above: serviceName.complexBehaviorName.stateName.action. "Body" property is created for viewing purpose with the format as "do /actionFileName.action"
- Use the creation tool "Final State" to add as many Final States into the Complex Behavior implementations as desired.
- Create the connection between the States by using the State Connection creation tool, drag from Initial State to the Intermediate State, and from Intermediate State to other Intermediate States or Final States. The implementation is considered valid if from the Initial State there is always a path to the Final States, and every States must be able to reach the Final States by following a certain path.
- Click on the connection and open the Properties View to see its own properties. Moving between States requires a State to satisfy a certain condition, this condition is specified in the Condition File property of the connection. All condition files are saved in the "conditions" folder in the workspace.

The resulting Complex Behavior implementation for Cloud Manager can be seen in Figure 6.5

The Complex Behavior implementation for SelfLet with Cloud Optimization Policy can be seen in Figure 6.6

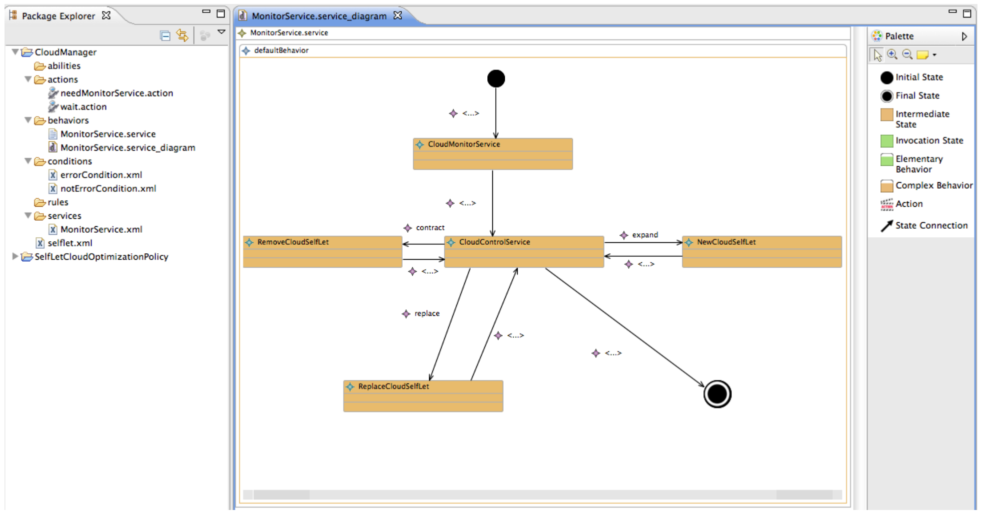


Figure 6.5: The Cloud Manager Complex Behavior implementation

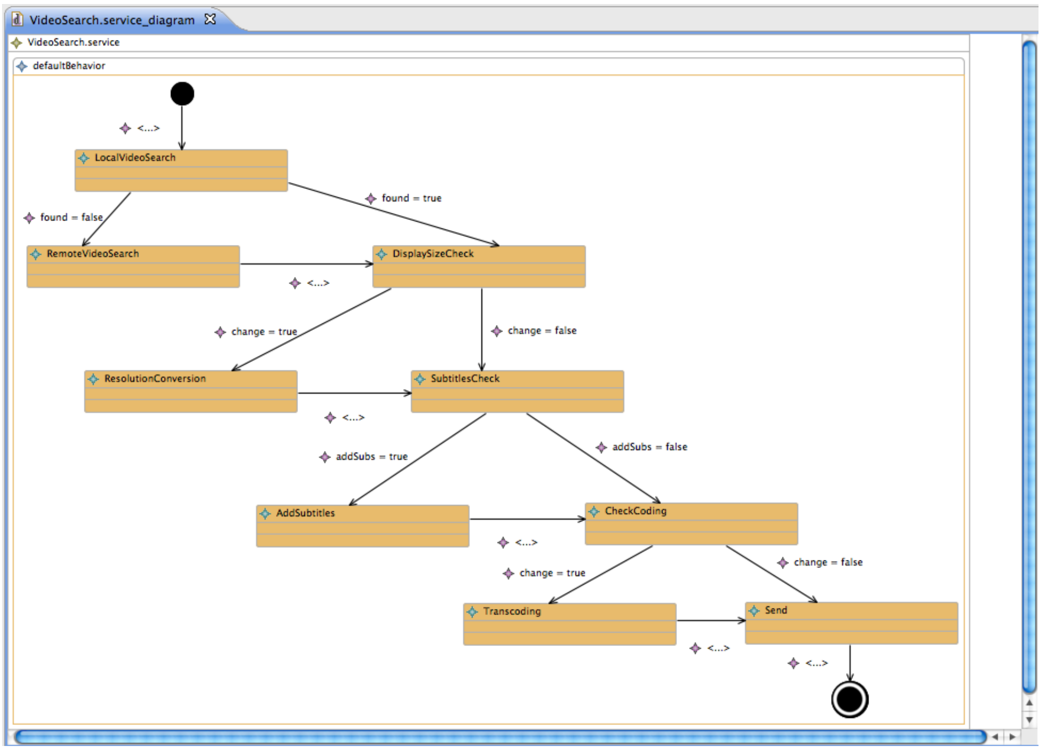


Figure 6.6: The SelfLet with Cloud Optimization Policy Complex Behavior implementation

The Complex Behavior implementation for Cloud SelfLet can be seen in Figure 6.7

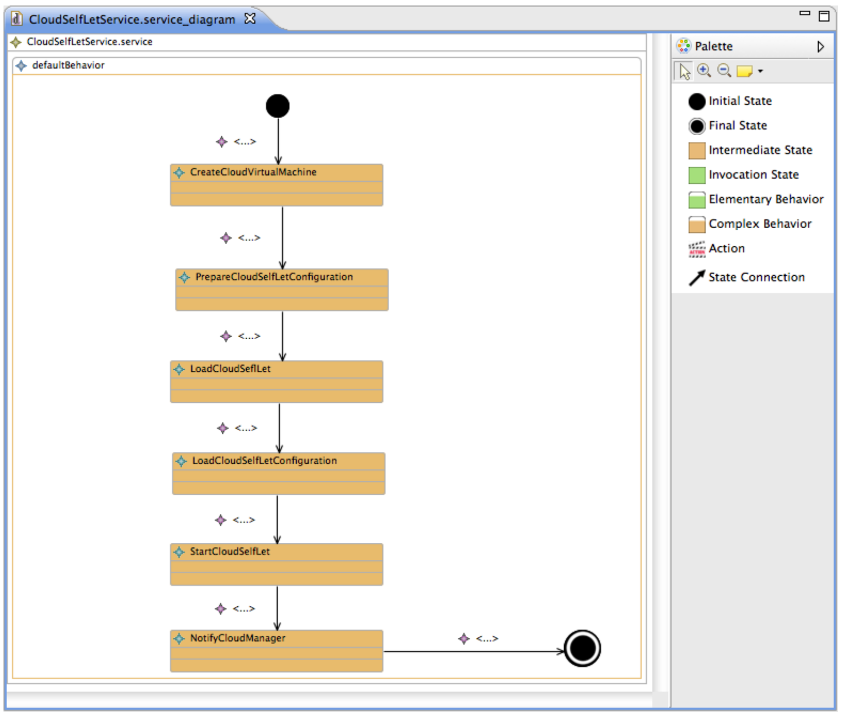


Figure 6.7: The Cloud SelfLet Complex Behavior implementation

6.4 Adding Elementary Behavior Implementation

`SelfLet` developers use the creation tool “Elementary Behavior” in the Palette to add Elementary Behavior implementation into a Service. This Elementary Behavior is positioned vertically inside the Service rectangle. Adding more Elementary Behavior is done by selecting the Elementary Behavior and clicking on the area outside of the existing Elementary Behaviors or Complex Behaviors of the Service.

Elementary Behavior has its Name displayed as a label and the property Name can be changed immediately when the Elementary Behavior is added into the diagram or with the properties View of the Eclipse Platform (Eclipse -> Window -> Show View -> Other -> Properties).

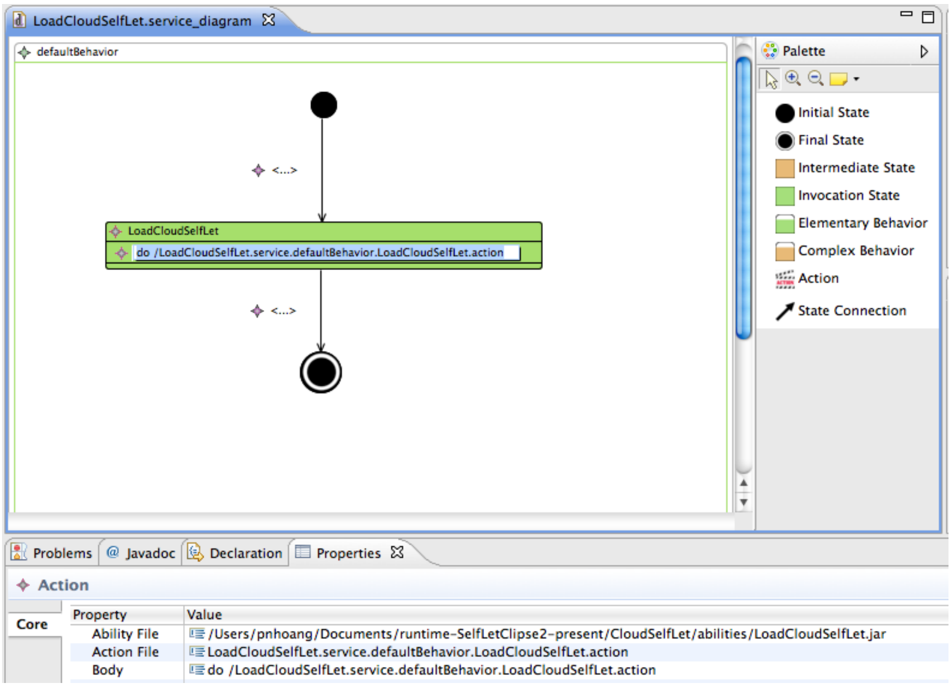
6.4.1 Adding States into the Elementary Behavior Implementation

In designing of the `SelfLet`, we have chosen to describe the behaviors of any Service as the State diagram. Elementary Behavior inherits this feature. The following list shows the steps which can be used to add States into the Elementary Behavior:

- Use the creation tool “Initial State” to add the Initial State into the Elementary Behavior implementation.
- Use the creation tool “Invocation State” to add the Invocation State into the Elementary Behavior implementation. Developer needs to specify its Name, so this Name can be used to create the Action file path and Action file template. Only one Invocation State can be added into the Diagram, adding more Invocation State into the Elementary Behavior causes the `SelfLetClipse2` IDE to raise a problem into the current file. This problem can be seen by using the standard Eclipse Platform Problems view (Eclipse -> Window -> Show View -> Other -> Problems).
- Use the creation tool “Action” to add the actual Action into this Invocation State. Action does not have Name, but they have 3 properties.
- Open the Properties view of the Eclipse Platform, click on the current Action to see its properties. The property Ability File needs to be set so that the current Invocation State does some real ability.
- Click on the three dots button next to the Ability File field to choose the Abilities file with .jar extension. The dialog has been fixed to enable choosing only the .jar files.
- “Action file” property is created automatically by the `SelfLetClipse2` IDE when it detects the valid .jar file has been selected. The format of the Action file template created in the project workspace is the following: serviceName.elementaryBehaviorName.stateName.action. “Body” property is created for viewing purpose with the format as “do /actionFileName.action”

- Use the creation tool “Final State” to add one Final State into the Elementary Behavior implementation.
- Create the connection between the States by using the State Connection creation tool, drag from Initial State to the Invocation State, and from Invocation State to Final State.

The resulting Load Cloud SelfLet Elementary Behavior implementation of the Cloud SelfLet project can be seen in Figure 6.8



Property	Value
Ability File	/Users/pnhoang/Documents/runtime-SelfLetClipse2-present/CloudSelfLet/abilities/LoadCloudSelfLet.jar
Action File	LoadCloudSelfLet.service.defaultBehavior.LoadCloudSelfLet.action
Body	do /LoadCloudSelfLet.service.defaultBehavior.LoadCloudSelfLet.action

Figure 6.8: The Cloud SelfLet - Load Cloud SelfLet Elementary Behavior implementation with its properties view

Other elementary behavior implementations for other Services and `SelfLets` are omitted as they have the same structures with the one described here.

6.5 Adding More Services

Adding new Service is done by going through the wizard `SelfLet` Service in the `SelfLet` category as we saw in Figure 6.1. The wizard takes the developers to the description page of the Service where developers can enter the Service Name and input, output parameters.

On finishing the wizard, three files are created in the project workspace: Service description file in "services" folder, Service Diagram and Service Diagram Model file in "behaviors".

The Service Diagram created has the empty Service container for storing Behavior Implementations in the State Diagrams. The name of the Service is given as default as the name of the Service domain model file.

6.6 Adding More Service Diagrams

Additional Service Diagram can also be added to the project in the workspace without having to create a new Service description. This is done by using the wizard `SelfLet` Service Diagram in Figure 6.1. This wizard asks the developers only the name for the Service domain model data file and Service diagram file names.

The result is the Service Diagram and Service Diagram Model file with the Service having its name as the name of the Service Diagram Model file.

6.7 The project structure and packaging the `SelfLet`

In this implementation of the `SelfLetClipse2` IDE, we have tried to make it easier for `SelfLet` developers to add their own abilities .jar files by providing the File Selector in the properties of the Action in the Invocation State of Elementary Behavior. Action file is created automatically as the templates for assisting `SelfLet` developers to add their own implementations. All the files are then saved into the file system within the project workspace. This folder structure might be packaged in a .zip folder as a self-managing package to deploy in the execution environment.

6.8 Conclusion

At this point, we have reached the final phase of creating three different `SelfLets` projects using the new `SelfLetClipse2` IDE available as the plugin for Eclipse. The resulting graphical editor allows the `SelfLet` developers to create their own fantastic graphical diagrams for `SelfLet` behavior implementations, the diagram domain data is saved in parallel in XML makes it possible to handle processing of the implementations by the Behavior Execution Manager and other Execution Environment in the `SelfLet` Framework.

In the next Chapter, we will make important conclusion and suggestions for future works of the project.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

In this thesis work, we have reached the final goal of implementing a new integrated development environment for the `SelfLet` framework within the context of autonomic computing. We would like now to summarize the important points in this work:

- In the first Chapter, we have briefly introduced autonomous computing concepts and how the `SelfLet` framework is related to this context.
- The internal structure of the `SelfLet` autonomous element is examined in details in the next Chapter, following by the conceptual model and the previous implementation of the `SelfLet`, the `SelfLetClipse` version 1.0 that supports three different wizards and context checks in the standard Eclipse Runtime Platform.
- In the next Chapter, we have presented the many new features of the new editor for the `SelfLet` environment, the `SelfLetClipse2`, in the heart of the editor is the graphical editor with the capability of creating stunning graphical diagrams suitable for Elementary Behavior implementations and Complex Behavior implementations of one or many Services in a `SelfLet` project. This graphical editor comes with a Palette for selecting the proper creation tools to draw into the diagram canvas, the Properties View is used as a method for `SelfLet` developers to add extra attributes and ability/action files into the `SelfLet` project in the workspace. In the

same Chapter, we have walked through a standard SelfLet development lifecycle.

- In the Chapter 4 we have introduced the meta-model for the graphical editor that we used for the implementation. This meta-model enables us to create the valid EMF editor for the domain data model. Having this EMF editor is the very important part of building the graphical editor because it ensures the correctness of the graphical editor on the domain data created.
- Chapter 5 walks through the remaining steps in creating the graphical editor for the SelfLet Service' Behavior implementations. Following the steps as described, one can easily make modifications to any part of the diagram from the Palette, to the Properties View, to the graphical representation of the elements in the graph. This enables great customization and enhancements for future improvements.
- The Chapter 6 has gone through a complete example of making a SelfLet project with the graphical diagram using the SelfLetClipse2 wizards and diagram palette.

7.2 Future Works

Due to the time limit, in this thesis work, an evaluation of how the SelfLet developers use the new SelfLetClipse2 to create their preferred SelfLet projects has not been taken. This evaluation must be done carefully in order to create the user-friendly graphical editor and integrated development environment for the SelfLet developers. One of the possibilities from this evaluation is the validity checks for the SelfLet being developed such as syntax checking, missing behaviors, checks on conditions, supports for autonomous policies that are written in Drools language, etc...

Another possibility is the support for the deployment of the newly created SelfLets. The developers must be able to design and run the SelfLets easily from the same interface.

Other improvement for the SelfLetClipse2 IDE might be the integration of Xtext[15] as the primary editor for the SelfLet configuration file, SelfLet developers then use this editor to insert autonomous elements into the `SelfLet` project. This requires combining Xtext with the three frameworks used in this thesis work: EMF, GEF and GMF for the complete integration between the Xtext editor and the graphical editor. All changes made in the Xtext editor should be reflected in the graphical editor and vice versa. The new process for creating the `SelfLet` project would be:

- `SelfLet` developers use the wizard to create the standard `SelfLet` project as in `SelfLetClipse2`. At the end of the project creation, the project folder structure is created in the workspace with the default diagram and the Xtext editor for the `SelfLet` configuration.
- `SelfLet` developers can either use the Xtext editor to add autonomous elements into the project or use the wizards to accomplish the same thing. The Xtext editor contains a parser/compiler for analyzing elements in the configuration and adds/removes the corresponding elements to/from the `SelfLet` project.
- `SelfLet` developers use the `SelfLetClipse2` graphical editor to add the Service Behavior Implementations diagrams.

Bibliography

[1] S. Bindelli, E. Di Nitto, R. Mirandola, and R. Tedesco. Building autonomic components: The `SelfLets` approach. *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference*, pages 17 – 24, Sept 2008.

[2] N. M. Calcavecchia, D. Ardagna, and E. Di Nitto. *The Emergence of Load Balancing in Distributed Systems: the SelfLet Approach.* Springer Basel, 2010.

[3] N. M. Calcavecchia, D. Ardagna, E. Di Nitto, and A. Gandini. Developing applications in the cloud through the selflet framework. Politecnico di Milano, Italy, 2011.

[4] N. M. Calcavecchia and E. Di Nitto. Incorporating prediction models in the selflet framework: a plugin approach. *1st International Workshop on Run-time mOdels for Self-managing Systems and Applications. Pisa, Italy*, 2009.

[5] N. M. Calcavecchia, E. Di Nitto, D. J. Dubois, C. Ghezzi, V. Mazza, and M. Rossi. *Complex Autonomic Systems for Networked Enterprises: Mechanisms, Solutions and Design Approaches.* Adaptive infRasTructures for DECentralized Organizations, 2011.

[6] Scott Chacon. Pro git, (`http://progit.org/`).

[7] JBoss Community. Drools the business logic integration platform, (`http://www.jboss.org/drools`).

[8] Software Freedom Conservancy. Argouml modeling tool, (`http://argouml.tigris.org/`).

[9] The Eclipse foundation. Eclipse editing framework, (`http://www.eclipse.org/modeling/gef/`).

[10] The Eclipse Foundation. Eclipse helios sr2 packages, (`http://www.eclipse.org/downloads/packages/release/helios/sr2/`).

[11] The Eclipse Foundation. Eclipse indigo package, (`http://www.eclipse.org/downloads/`).

[12] The Eclipse foundation. Eclipse modeling framework, (`http://www.eclipse.org/modeling/emf/`).

[13] The Eclipse foundation. Graphical modeling project, (`http://www.eclipse.org/modeling/gmp/`).

[14] The Eclipse Foundation. Rich client platform, (`http://www.eclipse.org/home/categories/rcp.php`).

[15] The Eclipse Foundation. Xtext, (`http://www.eclipse.org/Xtext/`).

[16] GitHub. Github social coding, (`https://github.com/`).

[17] Object Management Group. The object constraint language, (`http://www.omg.org/spec/OCL/2.0/`).

[18] Markus C. Huebscher and Julie A. Mccann. A survey of autonomic computing - degrees, models and applications. Imperial College London.

[19] J.O Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer Society*, 36:41 – 50, Jan 2003.

[20] Ngoc Hoang Pham. The selfletclipse2 editor on github, (`https://github.com/pnhoang/Selflet`).

[21] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, Second Edition.* Addison-Wesley Professional, December 16, 2008.

[22] Zhenxing Zhao, Congying Gao, and Fu Duan;. A survey on autonomic computing research. *Computational Intelligence and Industrial Applications, 2009. PACIIA 2009. Asia-Pacific Conference*, pages 288 – 291, Nov 2009.