Behaviour in Model-based Software Engineering:

A flexible coordination language based on AMFIBIA

Author: Yang Li

Supervisor: Ekkart Kindler

March 21, 2011
# Contents

1 Introduction  
  1.1 Example ............................................. 2  
  1.1.1 Event ............................................. 3  
  1.1.2 Interaction ....................................... 4  
  1.2 Problem Description  
    1.2.1 Interaction Cardinality ....................... 4  
    1.2.2 Joint Interaction ............................. 5  
    1.2.3 Event Parameters ............................. 7  
  1.3 Project Goal ....................................... 7  
  1.4 Thesis Structure ................................... 8

2 Chosen Technology  
  2.1 Eclipse Modelling Framework ........................ 9  
  2.2 Running Status of the Project ..................... 10

3 Concept Analysis and Discussion  
  3.1 Inter-Object Behaviour and Local Behaviour ......... 13  
    3.1.1 Local Behaviour ............................... 13  
    3.1.2 Inter-Object Behaviour ....................... 15  
  3.2 Interaction Fragments at Runtime .................. 22  
  3.3 Summary ........................................... 24

4 Notation of the Coordination Language  
  4.1 Meta-Model for Modelling Notations ................. 25  
    4.1.1 Meta-Model for System ....................... 25  
    4.1.2 Meta-Model for Local Behaviour .............. 29  
    4.1.3 Meta-Model for Defining Expression .......... 31  
  4.2 Meta-Model for Runtime Notations .................. 31  
    4.2.1 Meta-Model for Runtime Instance of a System 33  
    4.2.2 Meta-Model for Value ......................... 33  
  4.3 Summary ........................................... 35
Abstract

Model-based Software Engineering is a popular approach for developing software from models supported by automatic code generation. Though this is applied successfully in many cases, there are still some problems when it comes to modelling the actual behaviour of systems. One of the main challenges is integrating different kinds of behaviour models and integrating behaviour models with existing code.

The AMFIBIA approach showed that it is possible to integrate the behaviour of different parts of software by identifying events and combining these events into interactions, where the possible interactions are defined by a kind of coordination language. It turned out, that the concepts of AMFIBIA were powerful enough to model a workflow management system and some other applications. But, the concepts of AMFIBIA still lack some expressive power for modelling other kinds of systems.

In this project, the concepts of AMFIBIA was generalized and extended so that the coordination language becomes more flexible and can be applied in other application areas. These extensions, including cardinality labels, joint interactions and event parameters, have been designed carefully after a detailed discussion. New cardinality label 'some' has been introduced. Joint interactions on a same event are supported at runtime. Event parameter has been introduced and moreover, the mechanism of initializing the parameters has been set up.

As a proof of concept, a prototype of an execution engine for the proposed notation has also been implemented and demonstrated by an application example.
Preface and Acknowledgment

This thesis was prepared at the Department of Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the MSc degree in Computer Science and Engineering.

It presents the result of work carried out in the period from September 2010 to March 2011, with a workload equivalent to 30 ECTS credits.

I would like to express my deepest gratitude to my supervisor Ekkart Kindler, who has given me best guidance and support over the last six months. I appreciate for his contributions of time and ideas. He taught me a lot on both scientific knowledge and writing skills, for which I am deeply grateful.

I would like to say thanks to my dearest wife, for her encouragements and love during my master study at DTU.

Lastly, many thanks to my parents for their love and support.

Lyngby March 21, 2011
Yang Li
Chapter 1

Introduction

Model-based Software Engineering is a popular approach for developing software from models, modeling and model transformation. This approach has been widely discussed in software industry recently because it shows significant advantage in increasing the quality, efficiency and predictability of large-scale software development [1]. By supporting code generation, Model-based Software Engineering can reduce unnecessary artifacts (e.g. the generated program structure will be cleaner, consistent and produced very fast) during the process of software development. Therefore Model-based Software Engineering offers organizations tangible productivity improvements over previous software development approaches [2]. Though Model-based Software Engineering has been applied in many applications, there are still some open issues when it comes to modelling the actual behaviour of systems. One of the main challenges is integrating different kinds of behaviour models and integrating behaviour models with existing code [4].

AMFIBIA, is a meta-model that formalize the essential aspects and concepts of business process modelling in a formalism-independent manner [4]. It attempts to capture the interaction among the different aspects and concepts. In this manner, AMFIBIA is a possible approach to integrate the behaviour of different parts and different aspects of the software. By identifying events and combining these events into interactions, where the possible interactions are defined by a kind of coordination language, it turns out that the concept of AMFIBIA is powerful enough to model a workflow management system and some other applications. However, the concept of interaction in AMFIBIA is restricted so that it is efficient and appropriate to use[5]; yet it may need more expressiveness in some cases.

In some related research, e.g. MODOWA, an acronym for modelling domains with aspects, developed a modelling notation for aspect-oriented modelling and a prototype of a tool to execute these models as a proof of concept[6]. Its actual purpose is to model the interaction of behaviour of different aspects and execute
the coordination using the prototype. The coordination language of MODOWA puts its focus on coordinating the behaviour of different aspects. Inspired by the coordination language of MODOWA, it is possible to generalise the concept of AMFIBIA for the dynamic behaviour within a single aspect.

This project aims to develop some new concepts and notations for the coordination language to complement the original framework of AMFIBIA. Conceptually, the coordination language should be expressive and flexible enough to define various interactions in a simple and clear way. Nonetheless, it is very important to keep the language adequate, efficient and executable to use. Expressiveness and efficiency of the language need to be balanced during of the project.

Before going into the details of the solution, it is very important to clarify the scope of what the project concerns. Therefore, in this chapter, some key concepts of AMFIBIA will be introduced by an example first. Then follows the discussion of the problems with the current framework and the parallel work from other researchers. Afterwards, this chapter ends with a discussion of the project goal and a short introduction about the thesis structure.

1.1 Example

In this section, a simple web game example, will be presented. Demonstrated by the activities of the example, the key concepts of AMFIBIA, event and interaction will be explained in detail.

This example is a simple web game, in which players can move around on maps and fight with monsters within the same map. In runtime, the players and monsters are located on a set of maps. The players and monsters on the same map can move around the map to seek each other. Once they are close to each other (judged by their physical position), they can fight each other. There are two kinds of players, warrior and magician. On one hand, a magician can attack several monsters at a time using his magic power while a warrior can only fight a single monster at a time. On the other hand, a monster can be attacked by many players, no matter they are magicians or warriors.

The structural layout of the example web game can be expressed by figure 1.1. Notice that this figure looks like a class diagram in UML, but actually it is not. The note at the top left corner specifies the event types of a system. In this example, there are two different types of events, move and fight. The association classes in the figure, Move, PhysicalFight and MagicalFight denote the interactions of the web game. Elements of an interaction are listed in the class. The concept of event and interaction will be introduced in the following sections. A derived association close connects abstract class Player to class Monster. This association shows that the runtime links of the association will
be established when the condition is met, that two objects are positionally close to each other.

Figure 1.1: Model of Web Game

In another word, figure 1.1 is part of the concrete syntax of the proposed coordination language. However, since a graphical editor for the concrete syntax of the coordination language had not been developed yet, this figure was drawn using a UML diagram, just to express the ideas.

1.1.1 Event

In AMFIBIA, when discussing and modelling behaviour, often the idea that some objects, components and units need to synchronize at some points needs to be expressed [5]. These points have a meaning in the domain and can be identified in a domain, which is very similar to the concept of dynamic join points from aspect-oriented programming [3]. These points are often the start or the end of an activity, thus can be identified as points in the behaviour in which different partners participate. AMFIBIA used the term *events* to make these points an explicit modelling concept.
In the example mentioned in the previous section, the activities move and fight could be identified as the events in the domain model of the game. In this project, events will be used for coordinating behaviours and passing parameters of different parts in the model. The discussion about how they are defined and how they can be used for synchronizing behaviour will come in the later chapters.

1.1.2 Interaction

In AMFIBIA, an interaction denotes that objects are synchronizing on an execution of an event via an association, rather than triggering each other [5]. For example, when a player wants to move on the map where he locates; meanwhile the map is ready to update the new position for the player (a map needs to maintain a mapping from a position to an object at the position), they can synchronize together and execute the event move, so that the player gets to a new position and the map updates the player’s position. If a player and a monster are close to each other, they can synchronize and execute the event fight. As a result, the player and the monster receive a damage from each other. Both of the mentioned behaviours are interactions. According to the description of the example (sect 1.1), the interaction between magician and monster differs from the interaction between warrior and monster, since the cardinality mapping are different. This is also shown in figure 1.1.

For describing the interactions, a coordination notation on top of the structural models of UML will be introduced. This notation defines the abstract syntax specifying how to coordinate the behaviours in an interaction, including interaction partners, synchronization event and interaction cardinality.

1.2 Problem Description

The coordination language in AMFIBIA is very simple and implicit (there is no domain model for the coordination language itself) so far which leads to a lot of open issues in coordinating the behaviours [5]. In the following section, some problems or open issues of the present work of AMFIBIA will be introduced. In order to make the description clear, some concrete scenarios in the example will be described so that it will give a better feeling to understand what the problems are.

1.2.1 Interaction Cardinality

In AMFIBIA, the interaction labels 1:1 and 1:all are defined to specify the cardinality in an interaction [5]. Label 1:1 states that a single instance of a class interacts with a single instance of another class via an association link. Label 1:all specifies that a single instance of class interacts with all the instances from
another class via links of an association. However, these two labels are far from
enough to express all the cardinality mappings in interaction.

For example, label 1:some, which is in-between 1:1 and 1:all, is definitely
one of the most interesting labels to expand. In a situation shown in figure
1.2, Magician D, E and Warrior G locate on the map Mountain. Suppose that
Magician D and E want to move to seek monsters together; Warrior G wants
to just stay at the current position. Since in figure 1.1 the cardinality of inter-
action Move was defined as 1:all, Warrior G would be forced to interact with
map Mountain together with Magician D and E. By defining interaction label
1:some, this kind of problems could be solved.

In some particular cases, even the label some:some is interesting to define a
cardinality of interaction. For example, imagine a situation that Magician E is
able to fight both Monster A and C, Magician D is able to fight both Monster
B and C. But Magician E may not like to, since Monster C is too powerful that
he could not afford the damage from it. So he may only choose to fight with
Monster A. The situation is shown using a UML object diagram in figure 1.3.
In this regard, an interaction label some:some will help to define the interaction
properly.

Nevertheless, to decide which entity would participate in a 1:some or some:some
interaction involves non-determinism, thus it is beyond the project’s concern.
But to enable the coordination language to hold the power to model interaction
using these cardinality labels and to support executing these interactions in the
execution engine is the main concern of the project.

1.2.2 Joint Interaction

Figure 1.4 shows a particular example of three objects from three different
classes. Monster A is able to interact with both Warrior G and Magician
E. According to current coordination language of AMFIBIA, Monster A could
either fight Warrior G or Magician E. It makes sense at times.

But in some cases, it will be good if the different interaction can be joint so that the three objects interacts together, e.g. Warrior G and Magician E attack Monster A at the same time. To deal with such a situation, the current work of AMFIBIA needs to introduce an artificial sequentialisation of such concurrent behaviour, e.g. Magician E attacks Monster A first and Warrior G attacks Monster A later on or the way round. However, if it is possible to introduce the concept of joint interaction into AMFIBIA, such kind of problems would not be problems any more.

Apparently not only the additional concepts and notations need to be defined, but also the rules how a joint interaction is formed, since the new coordination language should not be chaotic and unexecutable. For example, as long
as the synchronization events are the same, two different interactions can be joint, this could be a rule of forming joint interaction.

1.2.3 Event Parameters

Another question is what an event could comprise besides its name. In AMFIBIA and MODOWA, the coordination language does not define parameters for event explicitly so far (AMFIBIA did provide a way to access attributes of objects participating in an interaction, which is called implicit parameters [5]). However, to define parameters explicitly seems to be a more safe way to transfer data values. Because the way of implicit parameters doing is actually collecting attributes’ value over the complete system, it cannot guarantee that the accessing attribute is initialized already.

Therefore, an important extension is to define the notations which enable the coordination language to define event parameters explicitly. What’s more, to construct the notations for this feature is one thing, to define the rules of how to use the event parameters (e.g. who will provide the value for the parameter and who will use the value) is another thing. The design of the new notations needs extra care.

A typical application of event parameter in the web game example, is the damage that a Player and a Monster receive from each other via interaction PhysicalFight or MagicalFight. If the notations can support defining two parameters (e.g. p_power refers to the damage that a monster suffers; m_power refers to the damage that a player suffers ) for event fight, it will be very convenient to calculate the damage for both sides, Players and Monsters.

1.3 Project Goal

In previous section, three open issues which exist in the current framework of AMFIBIA are mentioned. This project aims to expand the coordination language of AMFIBIA. By defining the domain model for coordination language explicitly and adding in new notations, the proposed coordination language should be able to model a system in a simpler and clearer way. The models defined by the new notations, should be expressive enough to express the idea of how the behaviours are coordinated. Also the models should be able to include event parameters and define how the parameters’ values are initialized clearly. In this regards, the problems mentioned in the previous section will be conquered and the concepts of AMFIBIA will be generalized and extended.

By defining new notations for interaction cardinality labels, the coordination language becomes more flexible in coordinating the interactions at run time. Objects have more freedom to decide partners who they want to interact with, not necessarily all the instances nor only one single instance from the partner
class.

If the notation of joint interaction is introduced in the coordination language, objects can interact with each other more freely. Objects are not restricted to synchronize within a single interaction at a time thus can reduce unnecessary artifacts (e.g. artificial sequentialisation of concurrent behaviour).

A careful design of event parameter mechanism can introduce parameter feature in AMFIBIA, which makes the coordination language more powerful to apply in many application areas.

These extensions, however, need to be carefully designed so that the interactions are still executable. As a proof of concept, a prototype of execution engine for the proposed notation will be implemented and demonstrated by an application example.

1.4 Thesis Structure

This chapter has given an introduction to the background of the project and discussed some problems and extension points of the current work of AMFIBIA. The rest of the thesis is structured as follows.

Chapter 2: Chosen Technology introduces the technology Eclipse Modelling Framework which has been chosen to implement our solution. Chapter 3: Concept Analysis and Discussion analyses the concepts of AMFIBIA in detail and discusses the design problems of the new notations, finally concludes with some design decisions had been taken in the project. Chapter 4: Notation of the Coordination Language presents the proposed coordination language in detail. It concentrates on explaining why the notations are designed in this way accompanied with some examples in abstract and concrete syntax. Chapter 5: Execution Engine introduces the design of execution engine. It first presents the domain model of the execution engine prototype. Then follows the explanation of the algorithm which computes the interaction fragments. Chapter 6: Implementation discusses some implementation details, including some minor details, like the data structure to implementing the cardinality constraint, the optimisation of the algorithm and so on. Chapter 7: Evaluation discusses the achievements of the solution and the efficiency of the algorithm. Chapter 8: Conclusion concludes the whole project work with discussion on the project results and suggestions to the future work.
Chapter 2

Chosen Technology

In the previous chapter, some basic concepts of AMFIBIA and problems of current work have been discussed. Before analysing to the concepts and discuss the design problems, this chapter will make a short introduction on the chosen technology, Eclipse Modeling Framework (EMF) and try to formulate the reason why it has been chosen.

2.1 Eclipse Modelling Framework

Eclipse Modeling Framework (EMF) is an Eclipse-based modeling framework. It supports code generation based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to generate Java classes for the model (EMF core framework), and adapter classes that enable viewing and command-based editing of the model (EMF.Edit framework) [9]. Moreover, based on the edit classes, EMF can generate a basic editor. Users can define their own models using annotated Java, UML, XML documents, or modeling tools. Then they can be imported into EMF.

Rather than arguing why choosing EMF to develop this project, the answer provided in this section is quite weak to convince others. Since the project time is not long enough to evaluate all the platforms and technology, the reason is more about the convenience and long term consideration for the project itself. Since the original work of AMFIBIA had chosen EMF to implement, it is more convenient to continue the work from what they have done.[4]

By designing the coordination language as ECore models, the source code is able to automatically generated from the models. Moreover, the generated models can be imported into EMF in the runtime workbench and the provided editor can be used to manipulate the generated models, with all the editor code can be generated automatically as well.
Due to the time limit of the master thesis project, the editor for the concrete syntax of the proposed coordination language has not been developed yet. But it is believed that if the project carried on, this could be implemented by using Graphical Modeling Framework (GMF), which is also Eclipse-based and compatible with EMF.

For these reasons, this project used EMF to implement all the features.

2.2 Running Status of the Project

Ultimately, a meta-model for modelling notations and a meta-model for runtime notations will be developed based on ECore model from Eclipse Modelling Framework. Some part of the meta-model for modelling notations actually extends the Ecore model directly, which will be discussed in Chapter 4. The meta-model for modelling notations actually defines the proposed coordination language.

One can use the coordination language to define a system model, just like a structure diagram in UML. Afterwards, he can also use the notations from the runtime notation meta-model to define a runtime model, which is an instance of the system model defined before. ECore, the meta-model of the proposed coordination language, the system model defined by the coordination language and the runtime model form a four-layered architecture, which is shown in figure 2.1.

These four layers have shown the running status of the project. However, another question is who is going to run the runtime models to coordinate the behaviours based on the new notations.

The answer is the execution engine. The execution engine runs an algorithm to compute which objects are able to interact together at runtime. It also maintains all the automatons of each object, fires transitions whose condition is met and perform transition activities. Finally it can be integrated with a GUI so that a user can directly manipulate on the models, e.g. fire transitions or trigger interactions.
Figure 2.1: Four-Layered Architecture of the Project
Chapter 3

Concept Analysis and Discussion

In the previous chapter, some basic concepts like event and interaction of AM-FIBIA were briefly introduced. These concepts gave the reader an overall picture of what the project concerns, yet they are not well explained in detail.

This chapter starts from explaining all the concepts involved in this project in detail accompanied with short examples. Then follows the analysis and discussion about how to design the notation so that it can somehow solve the problems mentioned in the previous chapter. Finally it ends with a summary about what design decisions had been taken during the project.

3.1 Inter-Object Behaviour and Local Behaviour

When modelling behaviours, the first issue is to classify whether the behaviour belongs to inter-object behaviour or local(intra-object) behaviour. Local behaviour refers to the behaviour those happen within an object while Inter-object behaviour refers to the behaviour which involves synchronizations between objects and components [7].

3.1.1 Local Behaviour

This section will first present the concepts of modelling local behaviour, leaving the concepts of modelling inter-object behaviour to be introduced in next section.

In principle, there are a lot formalisms to model local behaviour, e.g. UML has state machine diagram to describe the behaviour of a single object across several use cases [8]. State machine diagram from UML also supports internal
activities, concurrent states and some other notions, which is not this project’s concern. Therefore, this project has chosen automaton, which is a simplified version of state machine diagram, to define local behaviour. The same formalism was also used in AMFIBIA [5].

Automaton

Automaton, which is a simpler version of state machine diagram, uses finite states to identify different stages of a behaviour’s lifetime. Its transitions only support some basic elements like transition condition, synchronization event and transition operation. Figure 3.1 shows a simple example automaton for player fight behaviour in web game example from Sect 1.1.

![Automaton for Player Fight Behaviour](image)

Figure 3.1: Automaton for Player Fight Behaviour

The Idle state is the initial state of this automaton. A transition going out from state Idle to state Attack shows that the current object synchronizes with another object via event fight. Another transition going out from state Attack to state Idle with the activity ‘HP:=HP-m\_power’ means that the transition will perform an assignment subtracting the value of ‘m\_power’ from the attribute HP (HP is the health attribute of a PhysicalObject in Sect 1.1 and m\_power is a parameter of event fight which refers to the power of a monster). The last transition, which has no synchronization event and transition operation, has a condition ‘!HP>0’. It means that if the attribute HP’s value of the current object is no greater than 0, this transition will be fired. The transition will lead the automaton to the final state Dead. Having known the elements of a typical automaton, now the question is how to design the notation above it so that the notation could be used to define such an automaton.

There are many ways to design the meta model for automaton which is able to model an automaton like the one in figure 3.1. The basic layouts of
meta model from different ways are more or less the same. Usually the difference comes in expressing the initial state and final state. For example, David Schmelter had proposed a structure that states would be generalized into three sub classes, Initial, Simple and Final [6]. However, this design will lead to a problem when a modeler actually uses this notation to define an automaton. For example, an automaton has two states A and B with A as the initial state. Later on, when the modeler needs to change B as the initial state, he will have to reconstruct everything since A and B are two different classes (A is an Initial class and B is a Simple class. The modeler needs to delete the two classes and reconstruct an Initial class called B and a Simple class called A). This design makes the defined automaton almost uneditable. Another way is to use references to specify the initial state and final state, which is actually taken as the solution of the project. By changing the reference target, this design allows modeler to edit the automaton model easily. The detail design will be discussed in detail in Chapter 4.

After knowing the way how to define automatons, it is time to build up the relation between objects and their automatons. Actually the bridge is very straight and simple. In design time, a modeler could define some automatons within a class, which means that these automatons are the local behaviours of this class. The notation above this is a containment reference from Class to Automaton, which could be multiple. Details will be discussed in Chapter 4.

At runtime an instance of a class (object) can have those defined automaton instances running in parallel. For example, class Warrior from Sect 1.1, contains two automatons specifying the move and fight behaviour separately. Thus each Warrior object will have these two automaton instances running in parallel. Automatons can also be inherited from parent classes. For example, since both Monster and Player can perform move action, automaton to define move behaviour can be defined in PhysicalObject. Either a Warrior or a Magician should be able to run the move automaton to perform a move action.

3.1.2 Inter-Object Behaviour

As mentioned at the beginning of the section, the main feature of inter-object behaviour is synchronization between objects. In AMFIBIA, the concepts event and interaction are used to describe such kind of behaviour.

Interaction vs. Interaction Instance

Before talking about how to define the notations for interaction, one important concept needs to be clarified. That is the difference between interaction and interaction instance (interaction runtime instance).

Interaction is a kind of binary relationship between two classes from structure models, like UML class diagram. At the conceptual level, it defines two
classes, connected by a certain kind of association, are able to synchronize at a kind of event. At design time, a modeler can use this concept to construct synchronization relations for a system.

The concrete syntax for defining an iteration is shown in figure 3.2. The interaction \textit{MagicalFight} in web game example is composed of several elements, an association '/close', a cardinality label 'all:all' and an event type 'fight'.

![Figure 3.2: Definition of Interaction](image)

At runtime, an actual synchronization procedure is called interaction instance. When two objects are ready to synchronize, both of them must be first connected to each other by a link (an instance of the defined association in the interaction). Then each of them should have an automaton instance at a certain state that is waiting for the same event (more precisely event instance). Then they can synchronize when there is an event instance at runtime and perform the transition activities respectively.

An example to illustrate the difference between interaction and interaction instance is \textit{MagicalFight} in figure 1.1 is an interaction; \textit{Magician D} and \textit{Monster B} in figure 1.3 synchronizes as an interaction instance of \textit{MagicalFight}.

**Event vs. Event Instance**

Similar to the relation between interaction and interaction instance, the concept event is used at the conceptual level to define an action two classes are going to perform together; event instance is used at runtime to express a runtime instance of a type of event, which triggers an interaction instance.

**Interaction Cardinality Label and Definition**

As briefly introduced in Sect 1.2.1, introducing the notations to define the concept of 'some' for cardinality could be an extension for the project. This cardinality label 'some' means at runtime, the quantity of objects from a class
involved in an interaction interacting with objects from the other class involved in the same interaction could be from 1 to all. In another word, at runtime, if several objects are ready to interact with some other objects; not necessarily all of them should perform the interaction instances.

However, to introduce such a notation will lead to a problem in coordinate the interactions at runtime. Whether an object is able to participate in an interaction instance is one thing; whether it would like to do so is another thing. Especially when an object is able to interact with two other objects, but it has to choose one of the them to synchronize (e.g. constrained by the cardinality label ’1’). This problem will also occur if the concept ’some’ is introduced. To decide which object would like to participate in an interaction instance depends on users at runtime. Thus this problem seems to be an impossible mission for the execution engine.

What the execution engine could do is to compute all the possibilities for the non-deterministic objects and show that to users. By doing this, a user can select one possibility to execute. This could only work when the system’s object set is small enough at runtime. But as the scale of runtime system grows, the object set will grow really big cause the possibility set explodes. So this is definitely not worth to be implemented.

To completely solve this problem may need a long running research on non-determinism, which is impossible to perform in this master thesis project. There is another tricky solution for the execution engine. If a user concerns only the results showing the notations valid in coordinating interaction instances, rather than asks for a GUI to manipulate which objects to take part in an interaction, the execution engine is still able to use some specific algorithm (random select algorithm, mentioned in Sect 6.1.2) to coordinate the behaviours. This is actually what the project chose to do. So the project does develop a simplified version of ’some’ label under this circumstance. In fact, after adding the simplified version of ’some’ label, the proposed coordination language gains more power to model a system and the interaction cardinality within it properly and clearly. This is just the first step of using the ’some’ label to coordinate behaviour in AMFIBIA. Detail design will be left in Chapter 4.

The cardinality labels and their meaning extended by this project are shown in table 3.1.

### Joint Interaction

Consider the example mentioned in Sect 1.2.2. It is considered as advanced if two different interactions synchronizing on the same event can be joint. However, will it be more advanced to support joining interactions on different event?
CHAPTER 3. CONCEPT ANALYSIS AND DISCUSSION

<table>
<thead>
<tr>
<th>label</th>
<th>value in memory</th>
<th>actual meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Exactly one instance</td>
</tr>
<tr>
<td>all</td>
<td>-1</td>
<td>All ready instances</td>
</tr>
<tr>
<td>some</td>
<td>-2</td>
<td>Some ready instances</td>
</tr>
</tbody>
</table>

Table 3.1: Cardinality Label and Meaning

Imagine a situation that a Magician wants to move and fight with a Monster at the same time. It sounds very advanced if interaction MagicalFight and Move can be joint so that at runtime instances from class Magician, Monster and Map can synchronize together. This idea is shown in figure 3.3. On the left hand side of the figure is the structure model with notations defining the interactions; on the right hand side of the figure is a runtime model expressing that Magician D wants to interact with Monster A and Map Mountain at the same time. Unfortunately the current notation of AMFIBIA (the concept of joint interaction was not introduced in AMFIBIA [5]) does not allow him to do so, since interaction MagicalFight and Move are two different interactions.

However, to define such a notation to allow two different interactions can be joint on different event is easy, but to design an algorithm for the execution engine so that it can support computing the object set for joint interaction on different event at runtime is very complex. Moreover, the algorithm needs to be efficient enough. From a different perspective, using a sequence of interaction instances artificially can also solve the problem well. A Warrior can firstly move to a proper position and fight with a Monster. Because of the time limit of the project, the notation for joint interaction is defined with a constraint that only if the two interactions synchronize on a same event, they can be joint.

Imagine a situation that a Magician and a Warrior want to fight with the
same *Monster* together. On the left hand side of the figure 3.4 is the structure model with notations defining the interactions; on the right hand side of the figure is a runtime model expressing that *Magician* D and *Warrior* G want to interact with *Monster* A at the same time. Since interaction *MagicalFight* and *PhysicalFight* have the same synchronization event *fight*, they will be joint at runtime so that the three objects can interact together after the new notations are introduced into the coordination language.

![Figure 3.4: Joint Interaction on Same Event](image)

This could be even further extended if the event inheritance [5] feature is extended in the future. For example, two different interactions can be joint if the events they synchronize on two events sharing the same parent class. Detail design of the new notations will be found in Chapter 4.

### Defining Notations for Event Parameters

AMFIBIA has already got the notation to define events, but the notation allows an event only to have nothing more than a name. Having parameters to allow events to transfer data is very useful and thus can be considered as an important extension of AMFIBIA. However, the tricky part of the extension is not how to design the notation so that it defines the parameters explicitly. The most difficult and important part is to design the rules for assigning the values to the parameters. There are several ways to define the rules [5].

A simple way is to introduce an constraint that each parameter’s value is provided by one participant of the interaction. However, this constraint is too strict that it will just make the model too complicated in a certain circumstance. For example, if a modeler wants a parameter’s value to be the sum of an attribute from one participant of the interaction and a certain constant, it will be really hard to do this under this constraint (he may need to first sum up the attribute value with the constant and assign it to the event parameter and later
on subtract the constant from the attribute to revert the value of the attribute).

The other way is to allow more than one participant of the interaction contribute a value to the same parameter. A possible design is that the parameter’s value is required to be the same contributed from both participants of an interaction. This is called exclusive parameter. A typical example is that in electronic commerce, when a client submit an order, the server and the client usually agree on an auto-generated sequence code as the order number. This sequence code is typically an exclusive parameter in the event. This sounds good in the particular circumstances. However, it does not cover many other situations. A representative example is the parameter $p_{power}$ of event $fight$ from web game example in Sect 1.1, which could be a sum of Players’ power when a lot of Players attack the same Monster. The solution for dealing with this situation is to define a collective parameter, which is a collection of values. Those values are contributed from the participants of the interaction.

In order to implement a design which supports both exclusive and collective parameter, an important notation called ParameterValueProvider is introduced. A ParameterValueProvider is attached to a single event parameter. When a user uses these notations to define an event and its parameters, he is required to define a ParameterValueProvider for each parameters. Each ParameterValueProvider contains an expression to specify who will contribute to the value of this parameter. The expression could be an attribute from either of the participant, a parameter within the same event, arithmetic operations using attributes or parameters mentioned above, or even exclusive value specified from the external environment. Once this ParameterValueProvider is defined, the execution engine will be able to compute the values for the parameters at runtime. Figure 3.5 gives an example to define the events of web game.

Since the concrete syntax editor has not developed, this figure is drawn like a UML class diagram, but obviously it is not. The notation 'ExclusiveValue' for parameter $posX$ and $posY$ in event $move$ means the value of these parameters are provided by the external environment. In event $fight$, the notation 'Player.$p_{power}$' looks like Java’s syntax, which means the value of parameter $p_{power}$ is contributed by the attribute $p_{power}$ of class Player; the notation
3.1. INTER-OBJECT BEHAVIOUR AND LOCAL BEHAVIOUR

'Monster.m_power' is more or less the same meaning to express the value of parameter *m_power* is contributed by the attribute *m_power* of class *Monster*. Thanks to this design, whenever an interaction instance needs to be executed, e.g. an instance of *PhysicalFight*, the execution engine is able to initialize an instance of event *fight* and assign the values (values of the attributes from both participants of the interaction instance) to the parameters.

One may argue that this design is not flexible, since the event carries too much information. The notation of *ParameterValueProvider* could be defined in automats. But is it a good design? See the example in figure 3.6.

![Figure 3.6: ParameterValueProvider in Automaton vs. in Event](image)

In the figure, the automats on the left hand side define *ParameterValueProviders* in transitions. Suppose the upper automaton belongs to class A; the lower one belongs to class B. Both A and B have attributes a and b. From the figure, it can be seen that there are several problems if *ParameterValueProvider* is defined in automats. First, in a situation like this, which automaton should define the *ParameterValueProvider*? The one in A or the one in B? Or both of them? If only one of them defines the *ParameterValueProvider*, then it sounds more like the automaton which defines the *ParameterValueProvider* will initialize the event and event parameters, which triggers the other automaton. This conflicts with the concept interaction in AMFIBIA (interaction is not that an object sends an event to trigger another one). If both of them define the *ParameterValueProvider*, firstly it would cause redundant definition; secondly it will raise another problem which of the automats should be evaluated first to initialize the parameters? Therefore, the project chose a solution with less flexiblity. That is to define the *ParameterValueProvider* in events.

However, this design requires the modeler to do the modelling work very carefully, so that the *ParameterValueProvider* will not form a cycle in calculating the actual value of the parameter. For example, an event has two parameter A and B. A's *ParameterValueProvider* specifies that A’s value is equal to B’ value. In parallel, B’s *ParameterValueProvider* specifies that B’s value is equal to A’ value. Then a cycle is formed making the execution engine impossible to initialize the value for both of the two parameters. Besides, such a model cannot be considered as a appropriate model. Detail design will be discussed in Chapter 4.
3.2 Interaction Fragments at Runtime

When computing the available interaction instances among objects at runtime, another problem comes. Since an interaction instance is defined as a bilateral synchronization between two objects, it needs some new concepts to express the idea that several objects interact together, just like the case in joint interaction and so on. This notion, intertwining several interaction instances (sometimes could be just one interaction instance) together as one, is called interaction fragment. Interaction fragment refers to a set of interaction instances which can be considered as one and executed at the same time. What’s more, the involved objects of each interaction instances must be able to form a net structure, which means they are connected by existing links. This concept is quite abstract, therefore an example is provided to demonstrate the idea. Figure 3.7 shows a runtime model of the web game example, with all the objects are ready to synchronize on event \textit{fight}.

![Figure 3.7: Runtime Model of Web Game](image)

Obviously, if a graph is used to denote an runtime model, an interaction fragment is a subgraph of the runtime model. In this regard, there could be many ways to get a subgraph out of a graph. Figure 3.8 and figure 3.9 show two different results of computing the interaction fragments for the same runtime model. Figure 3.8 contains two interaction fragments while figure 3.9 contains only one interaction fragment.

Based on the notion of interaction fragment, the execution engine coordinates the interactions by computing available interaction fragments on a runtime model. The details about how the execution engine computes available interaction fragments will be mentioned in Chapter 5.
3.2. INTERACTION FRAGMENTS AT RUNTIME

Figure 3.8: Interaction Fragment Case One

Figure 3.9: Interaction Fragment Case Two
3.3 Summary

This chapter has explained some basic concepts and open issues of AMFIBIA in detail and discussed some possible ways to define new notations for these concepts and extensions. It is time to review some design decisions.

- Automaton is used to model local behaviour in this project.
- Due to limit time of the project, a simplified version of cardinality label 'some' is discussed.
- Regarding to joint interaction problem, a constraint has been set up that two interactions can be joint only when their synchronization event are the same.
- In order to extend event parameters, a notation called ParameterValueProvider is introduced.
- The concept interaction fragment is introduced for the execution engine to coordinate the interaction at runtime.

After these discussions, it is possible to go ahead to the solution.
Chapter 4

Notation of the Coordination Language

In Chapter 3, the design decisions of the relevant extensions are given after the discussion and analysis of the key concepts of AMFIBIA. This chapter presents the details of the proposed meta-model for the coordination language. In order to explain the meta-model clearly, the model has been split up into small parts and explained section by section.

4.1 Meta-Model for Modelling Notations

As mentioned in Sect 2.2, the project developed meta-models for modelling notations and runtime notations. This section will first present the meta-model for modelling notations, leaving the meta-model for runtime notations to be explained in next section.

The meta-model for modelling notations, which is used to define coordination language and model a system, comprises the meta-models for system and local behaviour.

4.1.1 Meta-Model for System

The meta-model for system defines how a system is composed and different parts (classes) of a system interact. The overall structure of the meta-model is shown in Figure 4.1.

The figure shows all the elements in the meta-model for defining a system model, which is an extension of ECore model and EMOF. A System consists of Classes, Associations, Events and Interactions. Class, which is generalised from EClass from ECore, is the main entity of a system.
Association, which has the same meaning with the notation in UML, is inherited from ENamedElement from Ecore model (In CMOF, association is generalized from Classifier. But since this project is using Ecore, which is actually EMOF with enhancement, it chose the definition from EMOF). In order to express a more dynamic way of interaction, the meta-model has defined two kinds of associations. ExplicitAssociation refers to the traditional association defined in UML. DescriptiveAssociation is a kind of special association similar to the notation derived association in UML, with some descriptions as conditions. These conditions will be defined using some simple expressions, which makes DescriptiveAssociation differentiated from derived association. The concrete syntax of the expression is defined in the sect 4.1.3. Each Association has a pair of Multiplicity, separately named as sourceMultiplicity and targetMultiplicity.

Event and Interaction express the respective concepts in AMFIBIA. Event defines the synchronization points of a system while Interaction defines the synchronization between the classes of a system. An Interaction comprises Event, Association, source and target Classes and Cardinality. The reason why Interaction keeps the source and target references to Class is to tell the concrete participant Classes (e.g. some interactions are based on an association connecting two super or abstract classes together. In this case, if the interaction does not keep the references to the concrete source and target classes, it cannot express the participant classes clearly). Cardinality is generalized from Mul-

Figure 4.1: System Meta-Model
4.1. META-MODEL FOR MODELLING NOTATIONS

Multiplicity, generally they are expressing the same concept. But to distinguish the multiplicity of association and the multiplicity of interaction, Cardinality is specially used to refer to the multiplicity of interaction.

Figure 4.2 shows the web game system model. The left hand side of the figure defined an abstract syntax of the web game in a tree structure editor. The right hand side of the figure shows the concrete syntax of the web game and how the interactions and events are defined.

![Figure 4.2: Web Game System Model Example](image)

**Interaction Cardinality Label**

It is the cardinality label that defines the cardinality mapping from source class to target class of the association and interaction. In reverse, the interactions will be coordinated based on the cardinality label. In figure 4.1, the cardinality label is implemented as an enumeration called `CardinalityLabel`. Notice that there are four values for the enumeration. Three of them correspond to the meaning defined in table 3.1. Value `zero` will not be used to define interaction cardinality, since it is meaningless to define an interaction in which no object will participate at runtime. However, since `Cardinality` is inherited from `Multiplicity`, Value `zero` will only be used in defining association multiplicity.
Joint Interaction

As described in Sect 3.1.2, the precondition of joint interaction supported by the proposed notation is two different interactions share the same event. Instead of defining the concept explicitly, it has been implicitly reflected by the model. The relevant design is shown in figure 4.3.

![Figure 4.3: Meta-Model that Defines Joint Interaction](image)

Notice that Class has a reference to Interaction specifying the interaction it could participate in. This reference design helps to find out what interactions have potential to be joint. For example, Warrior from web game could only participate in PhysicalFight and Move while Monster could participate in PhysicalFight, MagicalFight and Move. Since two interactions can be joint only if they synchronize on the same event, so all the interactions Warrior participates have no potential to be joint. Yet the interaction PhysicalFight and MagicalFight may be joint in runtime. The evaluation work for joint interaction is performed by the execution engine when it computes the interaction fragments. This will be mentioned in the algorithm design in Sect 5.2.

One may argue that why not define the Joint Interaction Concept explicitly. It is absolutely OK if it is done like that, but where to define this concept comes as an open issue. If it is defined within a Class, it will then be more or less close to the idea above, except that the two interactions are defined explicitly as part of joint interaction. This design will only increase the work load of evaluation for the execution engine. Imagine that a Class has many JointInteractions. When the execution engine tries to see whether it should join two interactions together, it needs to iterate through all the JoiningInteractions to see whether both of the two interactions are part of one of JointInteractions. That’s why the project does not choose to implement this concept in this way.

Event Parameter and ParameterValueProvider

Following the discussion in Sect 3.1.2, by using a containment reference from Event to Parameter, the notation allows to define parameters within events. Moreover, each Parameter contains a ParameterValueProvider, which is used to
initialize the value of the parameter. In order to explain the design more clearly, the meta-model defining the event Parameter and ParameterValueProvider has been extracted and shown in figure 4.4.

Figure 4.4: Event Parameter Meta-Model

Figure 4.5 shows the abstract syntax and concrete syntax for defining event and event parameters, including ParameterValueProviders. Notice that the concrete syntax of ParameterValueProvider has been discussed in Sect 3.1.2.

4.1.2 Meta-Model for Local Behaviour

Meta-model for local behaviour defines the elements of an Automaton. An Automaton is composed of Transitions and States. Transition has a reference to Event, which states via which event this object will synchronize with another
object at this transition. Another reference to TransitionOperation defines the activities to be performed (e.g. typically as assignments) if this transition is fired. The last reference to TransitionCondition specifies the condition of the transition. By using references to specify the initial and final states, the defined automaton is editable thus it is free from the problem mentioned in sect 3.1.1. Notice that the containment reference from Class to Automaton means a class owns the automaton. The proposed meta model is shown in figure 4.6.

![Automaton Meta-Model](image1.png)

**Figure 4.6: Automaton Meta-Model**

Figure 4.7 shows an example to define an automaton in abstract syntax and concrete syntax. Notice that some features like references to initial and final state are shown in Eclipse properties window thus cannot be displayed in the figure.

![Automaton Example](image2.png)

**Figure 4.7: An Automaton Example**
4.1.3 Meta-Model for Defining Expression

During the design of the meta-model for the coordination language, many concepts require notations to express the idea of expression. For example, Transitions may need arithmetic expression to manipulate the value of attributes or parameters as the condition or activities to perform.

An alternative solution is to use Java expressions. However, a parser needs to be designed and integrated with the existing notations of coordination language and the execution engine to be developed if the project chooses to use Java expression. This is beyond the concern of the project. Another solution is to design a totally new notation whose syntax could be very simple to define expressions. The advantage of this solution is easy to implement and integrate with the existing notations and the execution engine, since they are all developed using EMF. And the notation for expression is extendable during the development process of the project, thus it is very suitable for academic research. So the project chose to develop some new notations for expression.

The syntax of developed expression is simple, but it is still expressive enough to handle most business logic by supporting arithmetic operations. The expression meta-model is shown in figure 4.8.

Expression can be generalised as Variable, Constant and OperatorApplication. Variable, whose value is determined in runtime, is either an AttributeUsed or a ParameterUsed. Constant represents a constant value in the notation. OperatorApplication means that this slice of expression involved arithmetic operators, like BoolOperator, CompareOperator and NumericOperator. They all have an enumeration attribute referring to the corresponding operators.

Interface TransitionOperation, which refers to the activities to be performed when a transition of an automaton fires, typically is Assignment. MethodCall may be introduced in the future to model the concept of invoking a method. For an assignment, lefthand refers to a Variable while righthand refers to an expression. Similarly, TransitionCondition and DescriptionExpression has a reference to Expression to express the transition condition and condition of DescriptionExpression.

Figure 4.9 shows the abstract syntax and concrete syntax of an example.

4.2 Meta-Model for Runtime Notations

As a proof of concept, the execution engine should be able to show how the actual interactions happen(more precisely how the interaction instances are formed) in runtime. Therefore, the meta-model for runtime notations has been created and integrated with the meta-model for modelling notations in the previous section.


Figure 4.8: Expression Meta-Model

Figure 4.9: Expression Example
4.2. META-MODEL FOR RUNTIME NOTATIONS

4.2.1 Meta-Model for Runtime Instance of a System

Figure 4.10 shows the detail design of instance meta-model. In order to reduce the complexity of the figure, some details like operations and methods of the classes are not displayed in this figure.

![Instance Meta-Model](image)

Figure 4.10: Instance Meta-Model

Basically each class in meta-model for system has a corresponding runtime instance class in meta-model for runtime instance of a system, e.g. *Object* is a runtime instance of *Class*, *Link* is an instance of *Association* and so on. Notice that *EventRTInstance* denotes an event instance mentioned in Sect 3.1.2 *InteractionRTInstance* denotes an interaction instance mentioned in Sect 3.1.2. For demonstration, a web game runtime model is defined in figure 4.11, whose concrete syntax is the object diagram shown on the right.

4.2.2 Meta-Model for Value

After introducing the meta-model for runtime instance of a system, it is time to present the last meta-model, the meta-model for value. It defines the notations to express values of some certain datatypes. Since the execution engine is a prototype, it does not aim to support all the datatypes at the beginning. Four of the most popular datatypes are selected. They are *Boolean*, *Integer*, *Float* and *String*. The meta-model is shown in figure 4.12.

From the figure, *IntValue* generalized from *Value* comprise an *EInt* to express an instance of integer. Other classes have more or less the same structure.
Figure 4.11: A Runtime Instance Model of Web Game

Figure 4.12: Value Meta-Model
4.3 Summary

In this chapter, all the meta-models of the proposed coordination language have been discussed. However, for such a complicated meta-model, the overview of the whole meta-model and the connection or the dependency between models can hardly be shown here. Thus it can be found in the Appendix A.

Since the editor for concrete syntax of the proposed coordination language has not been developed yet, it is only possible to use this meta-model to model systems using abstract syntax. And because of this, the concrete syntax of the coordination language cannot be defined in detail. Nonetheless, the coordination language is not working until an execution engine is developed to practically show it works. Next chapter will introduce the detail design of the execution engine.
CHAPTER 4. NOTATION OF THE COORDINATION LANGUAGE
Chapter 5

Execution Engine

In the previous chapter, the meta-model for the coordination language has been presented. However, the actual coordination of interactions is performed by an execution engine as mentioned in Sect 2.2.

The execution engine, which actually maintains the information of a runtime instance model, is responsible for coordinating the interactions among objects and executing the interactions (including passing the event parameter, triggering automaton transitions and so on).

Moreover, the problem of how to design an execution engine is how to design the algorithm to compute the set of interaction fragments from a runtime instance model (Sect 3.2).

This chapter will first present the ECore model of the execution engine, giving an overall picture of the structure of the execution engine. Then it will focus on the explanation of the algorithm to compute interaction fragments.

5.1 Domain Model of Execution Engine

Since EMF supports code generation from ECore models, it is very fast and convenient to develop the execution engine using the approach of model-based software engineering. Using ECore models to define a domain model for the execution engine and afterwards generating the structure codes from the models saved a lot of time during the procedure of the project.

This section begins with the ECore model of the execution engine. Figure 5.1 shows the overall design of the execution engine. Notice that the elements in package instance are already defined in Sect 4.2.1.

Class InteractionFragmentPool is the core element of the execution engine, which is used to store interaction fragments. InteractionFragment is a class which refers the concept of interaction fragment mentioned in Sect 3.2. It con-
Figure 5.1: Execution Engine ECore Model
contains a list of InteractionRTInstance. The detail algorithm of how the execution engine compute the prospective interaction fragments will be explained in next section.

Another important element is AutomatonManager, which maintains all the automaton runtime instances for each objects. By evaluating the condition of transitions, it triggers the transitions whose conditions are met each time when there’s any changes happen on the Objects and AutomatonRTInstances(The changes are probably caused by executing an interaction).

Log helps to keep tracking of interaction executed so that during the execution, it is possible to know what is actually happening in the background.

5.2 Algorithm for Computing Interaction Fragments

Thanks to EMF, the structure codes for the execution engine can be generated automatically. The only thing to do is to implement an algorithm to compute interaction fragments manually. Before that, this section will first explain how the algorithm is designed and working.

Generally, consider a runtime model as a graph; an available interaction fragment is a subgraph of the runtime model at a particular time. Obviously there are a lot of possibilities to decide whether a single link belongs to this interaction fragment or that one even at this particular time. The algorithm implemented here does not aim to compute all the possibilities. Instead, it tries to provide only one possible ‘partition’ (not a real partition) grouping the links into different interaction fragments at a time. This ‘partition’ is not a real partition. It is based on the links of the graph (traditional partition is based on the nodes of the graph).

After introducing the concept of this special ‘partition’, a possible termination condition is that all the links have been owned by interaction fragments or ‘deleted’. If all the links of the runtime model are either assigned to an interaction fragment or ‘deleted’ from the runtime model, it can be claimed that at this particular time, a possible interaction fragment set has been computed. Of course the unselected links are not deleted.. They are just not selected to perform an interaction instance(due to the constraint of interaction cardinality). For convenience, the procedure to determine a link should be assigned to an interaction fragment or abandoned is named as being considered.

Be aware of the fact above, algorithm can be divided into following steps:

1. First of all, the algorithm marks all the links from a runtime model as ‘not considered’. Considering fast accessing, this step is done by using a
HashMap using Link as the key and a Boolean as the value. 'False' means this link has not been considered.

2. Then the algorithm starts with picking out an event type from the system. Notice that there's a constraint mentioned before that interactions can only be joint if they synchronize on the same event. Therefore, for a single interaction fragment, no matter there are joint interactions involved or not, all the objects inside this fragment can only synchronize on the same event. Based on this constraint, the algorithm starts with iterating the event types in the system.

3. Now it is time to iterate the objects from the runtime model. To avoid iterating an object twice, a HashMap using Object as the key and a Boolean as the value is used to express an object has been visited or not. 'False' means this object has not been visited.

4. For each event, a new interaction fragment is initialized (as mentioned before, it is a List of InteractionRTInstance). Then, the algorithm picks an object as a start point to start a recursion. The recursion is defined as follows. If this object is not visited before, mark it as visited. Then for all the links this object is connecting to, if a link has not been considered, and for all the interactions this object is able to take part in (remember the design of joint interaction mentioned in Sect 4.1.1), if there's one interaction's event matching the current iterating event and its association is the definition class of this non-considered link, it continues to next check. The next check is whether the both ends of the non-considered link (including the current object and its partner via this link) are ready to perform this event. If they are, it proceeds to check the cardinality constraints. If the cardinality constraint allows the interaction instance (it will be mentioned in next chapter) and the object would like to interact (this involved non-determinism, will be mentioned in next chapter), then add this link into the interaction fragment initialized before, go to the other end of the link (the current object’s partner) and do the recursion like mentioned here; otherwise if it is because the object would not like to interact, remark the link as 'not considered'; otherwise abandon this link (mark it as 'considered').

5. The recursion ends up with an interaction fragment containing some InteractionRTInstances. Notice that this interaction fragment will be triggered by the same type of event. Moreover, even for such kind of event, not all the links transferring this event are considered (some meet the condition but they don't want to interact in the previous step). It doesn't matter now the algorithm will go on with another type of event. But before the recursion, it will set all the objects as 'not visited' again.

6. After iterating all the events, several interaction fragment have been computed. If still not all the links are considered, the algorithm starts from iterating the first type of event again until all the links are considered.
5.3 Example to for Running the Proposed Algorithm

The following figures demonstrate the process of the algorithm presented in sect 5.2. Basically all the figures are the same object diagram of a web game instance. Each figure used different colours to mark the links and objects to illustrate a single step of iteration or recursion in the algorithm. Links with yellow colour refer to abandoned links due to cardinality constraint. Links with the same colour (except yellow) form an interaction fragment.

At the beginning when the algorithm starts. No links are considered in the object diagram.

Then the algorithm begins with iterating the event types of the system. Assume the first event type is fight. Object with a red cross states for the current object of the recursion. A red link is taken.
Then due to the recursion, the algorithm goes to the other end of the taken link and continue recursion. Objects with a red column stands for the visited objects.

The recursion continues and meets a link with cardinality constraint.
5.3. EXAMPLE TO FOR RUNNING THE PROPOSED ALGORITHM

Since all the links transferring fight event of the current object are iterated, the algorithm goes back to the previous recursive object and continues the recursion.

The recursion continues and meets a link with cardinality allows all:all; the link will be taken into the current interaction fragment for sure.
The recursion continues.

The recursion continues; all the links transferring event *fight* of the current object are iterated.
5.3. EXAMPLE TO FOR RUNNING THE PROPOSED ALGORITHM

The algorithm goes back to the previous object to continue recursion and meet a link with cardinality constraint.

For event *fight*, iteration ends with an interaction fragment is formed (all the links in red colour and the objects connected by them form an interaction fragment). For move event, some procedures are similar therefore they have been skipped here. Another interaction fragment is formed with the synchronization event *move* in green colour.
Since not all the links have been considered, the algorithm continues and starts from event *fight* again. The algorithm looks through all the objects which could synchronize with event *fight*, and found that all of them are already taken into an interaction fragment. Therefore no new interaction fragment can be found for event *fight*. Then the algorithm continues with event *move* again.

The algorithm continues until all the links are considered. Then the algorithm is finished with several interaction fragments.
5.4 Summary

This chapter has discussed the design of the execution engine, from the ECore model to the algorithm computing interaction fragments. During explaining the algorithm, some implementation details like the data structure storing the information are introduced. A detailed example to show how the algorithm works at runtime is provided. Next chapter will continue to introduce more implementation details.
Chapter 6

Implementation

After presenting the design of execution engine, finally comes the tricky part of implementation details.

This chapter will present some important details during the implementation of the project which have not been mentioned in the previous chapters.

6.1 Details in Computing Interaction Fragments

When the execution engine runs the algorithm designed in Sect 5.2, several detailed things needs to be implemented. First of all, the execution engine should be able to check whether an object is ready to synchronize on a specific event. Second, the engine needs to solved the ‘would like’ problem mentioned in Sect 3.1.2. Finally there must be some data structures to store the cardinality information for each object. All the problems will be introduced in the following sub sections.

6.1.1 Compute the Waiting Event for Objects

For each object at runtime, there is a data structure implemented manually to store which types of events the object is waiting to perform an interaction instance. This data structure is an ArrayList, which is called waitEvent.

To initialize this waitEvent, a method will go through all the Automaton-RTInstances of the owner object. For each AutomatonRTInstance, the method iterates all the transitions going out from the current state. If there is a synchronization event for a transition, add this event into waitEvent.

Later on, when the execution engine checks whether an object is ready to synchronize on a specific event, it will invoke this method and see whether the resulting waitEvent contains the event or not.
6.1.2 Randomly Select Strategy

For the problem to decide which object would like to participate in an interaction (mentioned in Sect 3.1.2), the project has chosen a randomly select strategy and implemented a random algorithm.

Class *FragmentPicker*, which encapsulates a static random selecting method, is implemented manually using a *Math.random* method to make the decision. If the random value is larger than 0.5, the method returns *true*, which means an object 'would like' to participate in an interaction instance; otherwise, even there's no cardinality constraint, an object won’t participate in an interaction.

Thanks to this algorithm, it is able to implemented a simplified version of cardinality label 'some' now. The execution engine is also able to coordinate interaction instances based on the cardinality labels, especially in the situation mentioned in Sect 3.1.2.

6.1.3 Data Structure for Cardinality

In Sect 1.2.1, the concepts of three cardinality labels are introduced. In Sect 4.1.1, the meta-model for cardinality labels are introduced. Now it is time to use a data structure to implement this feature, which enables the execution engine to check cardinality constraint at runtime. This data structure should exist in each object and may be accessed frequently to check whether there is any cardinality constraint for this owner object in participating this or that interaction instance.

Therefore an ideal candidate for this data structure is *HashMap*, which supports fast accessing. The solution is to use two *HashMap* using *Interaction* as the key and *Integer* as the value. They are called *sourceCardinality* and *targetCardinality*. When an object is initialized, the execution engine will scan all the interactions it can participate. If the object is the source of the interaction, the value of target cardinality of the interaction will be put into the *targetCardinality* map. Else if it is the target of the interaction, the value of source cardinality of the interaction will be put into the *sourceCardinality* map.

For example, when a *Warrior* object is initialized, the execution engine checks all the interactions it can participate in (Move and PhysicalFight). For interaction *Move*, *Warrior* is the source of the interaction, so the target cardinality value ’1’ (defined in the figure 1.1) is stored. Similarly, for interaction *PhysicalFight*, target cardinality value ’1’ is stored as well. The content of *HashMap* *targetCardinality* for a *Warrior* object is shown in table 3.1. Another *HashMap* *sourceCardinality* for a *Warrior* object is empty since for all the interactions a *Warrior* can participate, he is the source.

When the execution engine checks an object whether it has the cardinality constraint in an interaction, the engine will first check whether it is the source
### 6.2. Execution of Interaction Fragments

The execution engine runs an independent thread to compute interaction fragments and execute interaction fragments.

Table 6.1: Content of HashMap targetCardinality for a Warrior object

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>1</td>
</tr>
<tr>
<td>PhysicalFight</td>
<td>1</td>
</tr>
</tbody>
</table>

or the target of an interaction link. If it is the source, the execution engine will look up the value in HashMap `targetCardinality` using the interaction. If the value is -1, which means all instances (no constraint), then this object is able to participate in the interaction instance. If the value is 1, which means it can participate (not necessarily, but if it participates, it cannot participate in another interaction instance of the same kind), there are two possibilities. If it participates, the engine will decrease the map value to 0; otherwise leave the map value alone. If the value is 0, which means it has the cardinality constraint, this object is not able to participate in the interaction instance. The situation when the execution engine finds an object is the target of an interaction link is more or less the same, except the engine looks up the values in HashMap `sourceCardinality`.

At this point, remember the algorithm step 4 in Sect 5.2, the meaning of 'cardinality constraint allows' refers to the procedure of looking up the value in the two HashMaps. If the value is not '0', cardinality constraint allows the interaction instance to be executed; otherwise not.

#### 6.1.4 Algorithm for Evaluating Expression

To evaluate the value of an expression, an recursive algorithm is implemented. It requires an object and an event instance as parameters, which are required for evaluating attribute or event parameter values. The algorithm begins from the root element of the expression tree, check if the element is `OperatorApplication`, `Constant` or `Variable`. If it is `Variable` or `Constant`, it will return the value directly. Else if it is `OperatorApplication`, it will iterate through the supporting operators. After it knows which operator it actual is, the algorithm will go to its sub-elements to do the evaluation.

#### 6.2 Execution of Interaction Fragments

The execution engine runs an independent thread to compute interaction fragments and execute interaction fragments.

Thread `FragmentPoolManagerThread` is implemented to do this job. A main loop is defined in the `run` method of the thread. Basically the loop will first start with running the algorithm in Sect 5.2 to compute a set of interaction fragments and store them in the `InteractionFragmentPool`. Then it will randomly
select one of the interaction fragment to execute, which means it will execute all
the interaction instances within this selected interaction fragment. After that
the loop restarts over and over again until there is no new available interaction
fragments can be computed out. Then the loop throws an exception.

6.3 Details in Executing an Interaction Instance

When the execution engine executes a selected interaction fragment, several
detailed things needs to be implemented as well. First of all, the execution
engine should initialize event instances for each interaction instance as well
as their parameters. Second, the engine needs to fire the transitions, whose
synchronization event matches the coming event instance’s definition, of the
AutomatonRTInstance from the objects. These details will be introduced in the
following sub sections.

6.3.1 Initializing Parameters of Event

Thanks to the design of ParameterValueProvider and the algorithm for evalu-
ing expressions, the initialization for parameters becomes very easy to imple-
ment.

One thing needs to be mentioned is how to evaluate the value of the ex-
pression in ParameterValueProvider. Usually the objects involved in the Inter-
actionRTInstance which will be triggered by this EventRTInstance need to be
passed as parameters. This is because the runtime attributes’ value of these
objects may need to be accessed when evaluating the value of the expression in
ParameterValueProvider.

6.3.2 Firing Transitions in AutomatonRTInstance

To execute an interaction instance, the most important step is to fire the transi-
tions, which are waiting for the synchronization event instance, for each object
involved in this interaction instance.

Since the transition activity just supports Assignment at the moment, the
procedure of assign the value of an expression to the variable is very similar to
the initialization of parameter values. It requires the EventRTInstance as an
additional parameter to evaluate the expression since the expression may contain
the event parameter instances. Then the right hand side of the Assignment
could be evaluated and assigned to the left hand side (usually it is an attribute
of either of the objects in the InteractionRTInstance). The final step is to set
the current state as the target state of the transition.
6.4 Optimising the Algorithm

Every time after the `FragmentPoolManagerThread` executes an `InteractionFragment`, it needs to recompute the interaction fragment set again. If the object set is very large, this is very inefficient. Instead of recomputing the interaction fragment set for all the objects, the project implemented an algorithm which is smarter.

The principle is that the `FragmentPoolManagerThread` will only recompute the interaction fragment set for the affected objects (affected by the previous execution of interaction fragment) and the links in between.

The procedure for the optimization could be described as the following steps:

1. Select all the objects in the executed `InteractionFragment` and store them in an `ArrayList`. Then mark them as unvisited (using the `HashMap` mentioned in Sect 5.2).

Here is an example. Suppose links with different colours form an interaction fragment. There are three interaction fragments in the figure. Suppose the blue one has been chosen to execute.

![Diagram showing interaction fragments]

After execute the interaction fragment, there are still two interaction fragments in the `InteractionFragmentPool`. 
2. Iterate through the InteractionFragmentPool and delete all the InteractionFragments which contain one of objects in the previous executed InteractionFragment (using the ArrayList in the previous step).

After deleting the interaction fragments which contain one of objects in the previous executed InteractionFragment, there is only one interaction fragment in the InteractionFragmentPool.

3. Mark all the links which connect the objects in the ArrayList as 'not considered' (using the HashMap mentioned in Sect 5.2).
6.4. OPTIMISING THE ALGORITHM

The 'not considered' links and 'not visited' objects are marked with green colour in the following figure.

4. Finally use the new 'not considered' link set and 'not visited' object set to invoke the algorithm mentioned in Sect 5.2.

Two new interaction fragments are formed in purple and green colour.
6.5 Summary

This chapter has introduced some details when designing and implementing the execution engine and the model source code for the proposed coordination language. Now it is possible to run the execution engine and coordinate interactions at runtime based on the notations of the proposed coordination language.

Next step is to proof the usability of proposed coordination language and make a test on the execution engine to see the performance of the algorithm. This will be introduced in next chapter.
Chapter 7

Evaluation

In the previous chapters, the notations of the proposed coordination language were presented. What’s more, as a proof of concept, a prototype of execution engine has been developed. Now it is time to evaluate the work, to see whether the notations are able to solve the problems mentioned in chapter 1 and the execution engine can execute the interactions properly. Moreover, the notations should be adequate and executable to use.

This chapter will evaluate the project work from different angles, such as usability of the proposed notations, efficiency of the execution engine and integration with different kinds of behaviour models and existing code.

7.1 Usability of the Proposed Notations

In this project, there’s no time to actually put the developed notations into practice. Usually it requires a company to use the developed notations to develop several business projects and retrieves feedback from the engineers. Thus it is not possible to claim the usability of the developed notations right now. But as the execution engine shows, it is at least possible to say that the developed notations work theoretically.

7.2 Efficiency of the Algorithm of Execution Engine

The optimization of the algorithm to compute interaction fragments has been discussed in sect 6.4. However, the actual performance of the algorithm needs to examined to proof its efficiency.

Due to the thread design to execute interaction fragments of the execution engine, it is quite hard to perform a stress test on the execution engine. The
actual time to compute interaction fragment set cannot be examined easily. Besides the time frame of the project, the actual performance of the algorithm has not been tested.

### 7.3 Integration with Different Behaviour Models and Existing Code

#### 7.3.1 Integration with Other Behaviour Models

As mentioned in sect 3.1.1, this project has chosen automaton to model local behaviour; yet there are other behaviour models, e.g. activity diagram in UML. It is a promise at the beginning that the developed notations should have the possibility to integrate with different behaviour models.

Take the activity diagram as an example. The main advantage of activity diagram is it can support and show parallel behaviour clearly [8], which makes it a good tool to for work flow and process modelling. However, due to the time limit of this master thesis project, the developed meta-models do not support activity diagram right now. And if this project is carried on, it is possible to integrate with activity diagram for sure.

So the proposed coordination language is not able to integrate with other behaviour models at the moment.

#### 7.3.2 Integration with a Graphical User Interface

To proof the developed notations can integrate with existing codes, this project developed an independent graphical user interface. By invoking the interfaces from the meta-models, this GUI can run the execution engine and display the interaction information, which is more or less a kind of integration.

**Design of GUI**

Thanks to EMF notification mechanism, it is very fast to implement such a GUI using observer design pattern. The design of GUI is shown in Figure 7.1.

The GUI package, which encapsulates all the classes relevant to graphical user interface, is designed based on the EMF notification mechanism. Basically every entity in a system instance has a graphical output like `ObjectGUI`, `AutomatonGUI`, `AttributeGUI`, `EventGUI`, `ParameterGUI` and `LogGUI`. All these graphical classes have references to the relevant classes from instance package(sec 4.2.1) or `executionengine` package.
7.3. INTEGRATION WITH DIFFERENT BEHAVIOUR MODELS AND EXISTING CODE

At runtime, class GUI will initialize an ExecutionEngine. The ExecutionEngine will initialize the objects and links at runtime. Then GUI will call the GUI classes to invoke set method to set up their relevant references to the corresponding classes (e.g. an ObjectGUI will set a reference to an Object). Take ObjectGUI as example, the communication among GUI, ObjectGUI, Object and ExecutionEngine is shown in figure 7.2.

In the figure, the last send message ‘notifiedChanged’ is implemented by EMF. EMF notification mechanism requires the observer class (ObjectGUI) to initialize an adapter (like a listener) on the target class (Object). Thanks to this design pattern, the user interface will be very dynamic and the user can see all the changes happening at real time.

**View of GUI**

In previous section, the model of GUI and the relation with the execution engine has been discussed. When designing the graphical user interface (GUI) of the execution engine, this project chose to develop it as an Eclipse plug-in view project. In this case, the whole application, including the coordination language modelling editor and the execution engine can be run on Eclipse Run-Time Workbench. What’s more, the layout of GUI is dynamically generated according to the number of entities from a input system instance file.
Figure 7.2: Communication between GUI and ExecutionEngine
7.3. INTEGRATION WITH DIFFERENT BEHAVIOUR MODELS AND EXISTING CODE

Figure 7.3 shows a screen shot of Eclipse Run-Time Workbench and a possible picture of execution engine GUI.

On the left side, all the objects of the system will be shown with their attribute values and automaton states. Whenever an interaction is executed, the GUI will update the information of attribute values and automaton states if they are affected or changed. On the right hand side, the log records will be displayed so that the user can see what kind of interaction is executed and which objects has participated in the interaction. The fired transition will also be marked down and shown as a log record.
Figure 7.3: Graphical User Interface
Chapter 8

Conclusion

In the previous chapters, the thesis has presented all the works of the project. It is time to end with a conclusion.

8.1 Achievement

This project has developed meta-models for a new coordination language, which allows users to model systems and the interactions within the systems using the new notations. Since a concrete syntax editor has not developed due to time constraint, users could only use the abstract syntax of the coordination language.

The new coordination language, by introducing some new notations, has extended the original work of AMFIBIA and MODOWA. The new notations allow users to define new cardinality labels, joint interactions and event parameters, which give more flexible power to the coordination language in modelling.

During exploring solutions to define cardinality labels, it has been figured out that to decide which objects will participate in interaction instances involved non-determinism issues. As a result, the project has implemented a randomly select algorithm to solve non-determinism problems.

The proposed coordination language does not allow to define joint interactions on different events, due to the complexity of designing an algorithm to compute interaction fragments supporting that. Instead, the proposed coordination language allows to define joint interaction on the same event.

When defining event parameters, the proposed coordination language requires modelers to define a ParameterValueProvider for each parameter. This design helps to initialize the value for each parameter at runtime.

As a proof of concept, a prototype of execution engine has been developed
as well. An algorithm computing interaction fragments for runtime model has been developed. Moreover, the result of the algorithm shows that the execution engine is able to coordinate behaviours based on the new notations. Despite the performance of the algorithm to compute interaction fragment, the usability of the concepts has been proven.

The meta-models of the coordination language is an extension of ECore models which makes it possible to integrate with different behaviour models. A graphical user interface has also been provided integrating with the meta-models of the coordination language. It has somehow proven the developed notations can integrate with existing code.

Altogether, during the time of this master thesis project, the main goals are achieved. However, there is always a lot of space to improve the work.

8.2 Future Work

8.2.1 Concrete Syntax Editor

One of the main improvements is definitely to develop a concrete syntax editor for the proposed coordination language. Fortunately, Eclipse has provide GMF to implement this work. Once the concrete syntax editor is developed, more details of the concrete syntax of the proposed coordination language could be further discussed.

8.2.2 Usability of the Notation

As mentioned in Sect 7.1, the usability of the notations could only be proven until the proposed coordination language is actually put into practice. For long term consideration, this could only be done and further improved co-operated with software companies.
Appendix A

Meta-Models

A.1 Modelling Concept Meta-Model

Figure A.1: Modelling Concept Meta-Model
A.2 Runtime Concept Meta-Model

Figure A.2: Runtime Concept Meta-Model
Bibliography


[2] Steven Witkop. Driving business agility with Model Driven Architecture, Ohio/Pennsylvania Solution Centre, EDS.


BIBLIOGRAPHY