Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava

Code Generation from AML to Jadex

(Master's thesis)

Attila Mészáros

Thesis Advisor:
Mgr. Radovan Červenka, PhD.

Bratislava, 2010

Code Generation from AML to Jadex

Master's thesis

Attila Mészáros

Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava

9.2.1 Informatics

Thesis Advisor

Mgr. Radovan Červenka, PhD.

BRATISLAVA 2010

Hereby I declare that the work presented in this thesis is my own, written only by help of referenced literature.

Bratislava, May 2010

…..............................
Attila Mészáros

I would like to thank my thesis advisor Mgr. Radovan Červenka, PhD for guidance, advices and suggestions.

Abstract

Development and maintaining a large-scale multi-agent system is a very complex problem and a challenge for software engineering. Analysis, design, implementation, testing and maintenance of such systems could be very difficult to realize. Modeling of such systems in analysis and design is a crucial instrument to cope with complexity. In addition, automatic code generation simplifies the transition of analysis and design models to implementation artifacts, and therefore, can make development of software systems more effective.

Agent Modeling Language (AML) is a comprehensive agent-oriented modeling language that is used to capture the various aspects of multi-agent systems, therefore facilitates their development. In this work we implement a code generator that produces source code from AML models to an agent development framework, based on DBI software model, called Jadex. We define a mapping from AML to Jadex, and introduce a simple extension to AML that allows to generate high detailed source code. We provide implementation of code generation using Acceleo engine, following a very flexible approach that enables realization of code generation independently from CASE tools, using XMI files.

Keywords: Code generation, AML, Jadex, mapping, XMI.

Table of Contents

1 Introduction

In this section we give a short introduction to the context of our work. We give a more abstract view about technologies, methodologies that served as motivation for us. In addition we define the tasks and objectives of this thesis, and examine closely related works.

1.1 Model Driven Engineering

Building large scale systems is a complex and challenging problem in software engineering. A number of software development methodologies and techniques were introduced in the last decades in order to cope with designing, implementation, testing and maintaining such systems. These methodologies were used to facilitate, thus structure, plan and control the process of development. Modeling is a crucial aspect of most software development methods, models are used to describe different layers of abstraction of a system, and capturing essential and/or critical aspects of a system. It provides a mechanism that simplifies the whole process of software development. Models are developed through extensive communication among the product managers, designers and development teams, in this sense modeling also promotes the communication, and understanding of the system.

One of the well known families of software system development methodologies, that is based entirely on models, is Model Driven Engineering (MDE). It focuses on creating models and assists the whole process of software development. In these methodologies the whole process of software development can be interpreted as transformation of models from one to another, from model of requirement through models of analysis, design and implementation to model of testing. The best known initiative of MDE is Model Driven Architectures (MDA) [44], product of Object Management Group (OMG) [58]. MDA provides a set of guidelines for structuring specifications expressed as models, thus defines system functionality in platform-independent model (PIM) which is then translated to platform-specific models (PSM), that represent the system in the scope of the target platform where the system will be executed (MDA inspired us also from view of code generation, therefore it is described in more detailed way in section 1.4).

Model Driven Engineering became very popular with the introduction of the Unified Modeling Language (UML) [57], which as a general-purpose modeling language nowadays serves as a standard implemented by most Computer-Aided Software Engineering (CASE) tools. These applications implement features and functionalities that cover all aspects of development of software, i.e. analysis, design and programming. UML is a general-purpose language, so it is intended to capture every aspect of a software-system: structural, behavioral views and interactions of its components. In addition, it is designed to be extendible, UML Profiles provide a mechanism for customizing UML for a target domain or platform (see [56] for details).

1.2 Modeling Multi-Agent Systems

Models play an important role in developing of information systems in general, it is also a crucial part of multi-agent system (MAS) development. Development of a large and complex multi-agent system faces analogous challenges as development of an information system. However, in case of open multi-agent systems the situation can be even more complicated, as denoted in [18]. Problems can emerge like heterogeneity (agents may have different possibly inconsistent goals), communication problems (different communication protocols could be used by agents), and security problems (thus authentication and authorization of agents). To solve this problem an active research has been undertaken in this field in the last two decades in order to create languages, abstractions and methods and also toolkits that facilitate the whole development process of such systems. Results of this research formed the paradigm of Agent-Oriented Software Engineering (AOSE). Thus to cope with the specific features of agent-based system new modeling languages and methodologies were created, such as Gaia [14], Tropos [55], MAS-ML [23], AOR [54], etc. Another family of agent-oriented modeling languages is based on UML. These languages are implemented as extensions of the UML metamodel, such as AUML [53] or MESSAGE [21]. The Agent Modeling Language (AML) is also a member of this family, it is a semi-formal visual modeling language for specifying, modeling and documenting systems in terms of concepts drawn from MAS theory [18]. In comparison with other languages, AML covers a wide variety of aspects of multi-agent systems, is well documented, and is supported by modeling and other automation tools.

1.3 Overview of Agent Platforms

Agent-oriented approach facilitates the design of complex systems, because it gives a possibility to have the same concept, i. e. agent, as the central one, in the problem analysis and the solution design and implementation. Development of a software system with AOSE usually involves utilization of an agent middleware or platform. Agent platform is a technological architecture providing the environment in which agents can actively exist and operate to achieve their goal. An agent platform may additionally support the development of agents and agent based applications. In this section we provide a short overview of agent platforms and related terms.

One of the crucial aspects of agent frameworks are standards. To facilitate interoperability between platforms, and to specify how agents themselves should communicate and interact a set of standards was created. The most known standards are provided by Foundation for Intelligent Physical Agents (FIPA) [52]. This organization currently provides 25 specifications. A sub-set of these is already completed the process of standardisation. Other international standards exist like OMG MASIF [51] or Mobile Agent Facility [50], but FIPA is the most prominent one.

On of the best known example standard of FIPA is the Agent Communication Language (ACL) [49]. The purpose of this language is to make agents understand each other; they have to not only speak the same language, but also to have a common ontology.

In following we will introduce a few prominent agent frameworks:

- Java Agent Development Framework (JADE) [48] is one of the best known agent frameworks, that allows the development and coordination of multiple FIPA compliant agents. It uses the standard FIPA-ACL language. JADE is opensource, implemented in Java programming language.

- Jadex [2] is Java based, opensource, FIPA compliant, agent environment, that follows the BDI model. Jadex provides a framework and a set of development tools that facilitates the creation and testing of agents. Jadex is one of the central subjects of this thesis, we provide detailed overview of it in section 3.
- Grasshopper [47] is an open Java-based mobile intelligent agent platform. It includes two optional open source extensions providing the OMG MASIF and FIPA standard interfaces for agent/platform interoperability.
- Cougaar [46] is an example of not FIPA based agent platform, which is also an agent based distributed platform. It is a highly scalable framework implemented in Java.

There is a large number of commercial or not commercial agent platforms, however a complete overview or evaluation of them is out of the scope of this thesis. A list of available FIPA compliant agent platforms can be found at [45]. A comparison and performance evaluation can be found in [20].

1.4 Model-Driven Code Generation

The promise of modeling has been to shift the focus from implementation to design. Models serve as mechanisms to get a better understanding but they can also be an input for code generators. This automates development leading to improved productivity, quality and complexity hiding. The generator specifies how information is extracted from the models and transformed into code.

One of the currently best known and accepted code generation approach is based on Model Driven Architectures (MDA) [44]. In fact MDA defines a more abstract approach to development of system, it provides a set of guidelines for specifications, the specifications are expressed in model, usually in a high level modeling language like UML. MDA separates the business or application logic from the underlying platform. From the technical point of view it is related to set of standards, including Unified Modeling Language (UML) [56] , Meta-Object Facility (MOF) [43] , XML Metadata Interchange (XMI) [42] and the Common Warehouse Metamodel (CWM) [41]. These technologies are used to transform a model to engineering artifacts, in our case to source code. MDA defines some high level steps and how models are transformed, until with the last step of transformation the source code is generated.

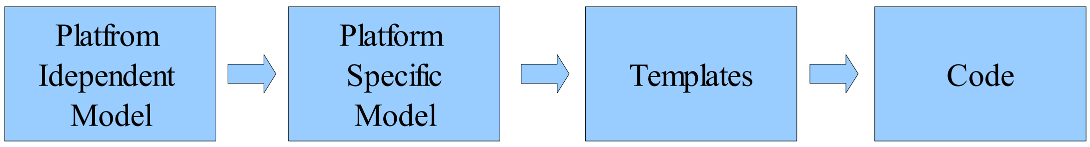

Figure 1: The MDA process

As illustrated on Figure 1, the Platform Independent Model (PIM) specifying system requirements, functionality and behavior undistorted by technological details is created as first. Then, a mapping is applied to generate a Platform Specific Model (PSM). This model captures in addition some details of the target platform. In other words the goal of this step and model is to bring closer the business requirement or the logic of the system to the target platform. In the last step we generate target source code using predefined templates. For additional information see [44].

MDA was one of the main sources of inspiration for code generation, however we will not strictly follow its guidelines.

1.5 Tasks and Objectives

The first goal of this thesis is to discover relations between AML and the Jadex agent platform. In our case this means that we will define a mapping of AML elements to artefacts of Jadex. Focus of our examination will be the Mental package of AML, which is used to model mental attitudes of autonomous entities, since the main and distinguishing characteristic of Jadex is that it implements a reasoning engine based on BDI software model (see section 3.1). In Jadex the mental aspects of an agent are explicitly described using Agent Description Files (ADF).

Our next goal is to specify an extension to AML that allows to describe the rest of principal aspects of Jadex system, which would be hard to capture using strict AML or standard UML elements. This enables to generate high detailed source code; focused on ADF files and other artefacts.

In the practical part of the work our goal is to implement a code generator that generates executable source code from AML models to Jadex platform. To carry out this task we will use a not so widespread approach, code generation from XMI files. We will also examine the possibilities and limits of this concept.

1.6 Related Work

In this section we examine the relations to a work which has a subject very similar to the subject of this thesis.

In his work Jiří Bělohlávek and Petr Knoth describe a translationn [7] between Prometheus [11] methodology and Jadex. However this work is rather a short demonstration of capabilities of Jadex for implementation of Multi Agent Systems designed using Prometheus Methodology.

Michal Kostič introduced his master's thesis [15] in 2006, with title “AML Code generation”. This work specifies a mapping between AML and agent platform Jade. The author also provides an implementation using CASE tool inner mechanisms, and explores some code generation related topics. The concepts of our and his work are close to each other. However, this thesis can be rather interpreted as a complementary work, in sense:

- in [15] the mapping is provided from Architecture and Behavior packages of AML. The author identifies members of Mental package as elements which are not suitable for code generation, what is reasonable in case of code generation to Jade. However, in our work we introduce code generation mostly from this package. There is no contradiction, we are enabled to do so by characteristics of Jadex as described in section 4,

- in addition we use a different approach to code generation. While Kostič in his work follows a CASE tool dependent approach, thus implements the code generator as an add-in, that uses the CASE tools API to access the model. Our implementation follows a different approach: it generates source code using the XMI file format (see section 6).

Therefore some parts of our work, like mapping in section 4, we describe in a similar form as was provided in [15].

1.7 Structure of the Document

The document is structured to 7 main sections. In the first section we give an introduction to the context of the work and related terms. In section 2 and 3 we give an overview of the systems that are the central subjects of this theses. We describe elements of Jadex and AML that are closely related to this work. These elements are used in section 4, where we introduce the theoretical part of our contribution. In this section we provide a mapping from AML to Jadex, which is then illustrated in section 5. Examples in section 5 clarifies how the mapping is used to generate source code from AML models to Jadex. In section 6 we give a description of our implementation, also an overview of the approach what is followed by our implementation. In section 7 we describe possible extensions or complementary solutions that are close to the subject-matter of this thesis.

2 Overview of AML

In this section we will provide an overview of Agent Modeling Language. We will give an introduction to the language, to its packages, and a more detailed description of AML Mental package, which is in the focus of our interest from the scope of code generation. For more detailed description refer to [17], or for formal specification see [6].

2.1 Introduction

The language specification [16] defines AML as: "The Agent Modeling Language (AML) is a semi-formal visual modeling language for specifying, modeling, and documenting systems in terms of concept drawn from MAS theory." As the definition says, the primary application context of AML are systems, which design principles are adopted from multi-agent systems. The scope of AML also facilitates business modeling, requirements specification, analysis and design of software systems, that uses MAS paradigm. The support for requirements specification and analysis of complex problems covers mental aspects, which can be used to model goal based requirements, problem decomposition, etc.. Also covers contexts, which are used for situation based modeling. Support of AML for abstraction of behavioral and architectural concepts covers topics like:

- social aspects,
- communicative interactions,
- services,
- behavioral abstraction and composition,
- or mental aspects.

2.2 Language Architecture of AML

AML is based on UML 2.0 superstructure, it is defined at two distinct levels:

- AML Metamodel and Notation – this level defines the AML abstract syntax, its semantic and notation. The metamodel is further also structured into two packages. First of them is the AML Kernel package, where the core language constructs are defined. This is a conservative extension of UML. The other package is UML Extension for AML, which adds meta-properties and structural constraints to the standard UML elements. This package is a non-conservative extension of UML, and its an optional package of AML
- AML Profiles – there are two UML profiles defined, one for UML 1.* and an other for UML 2.0.

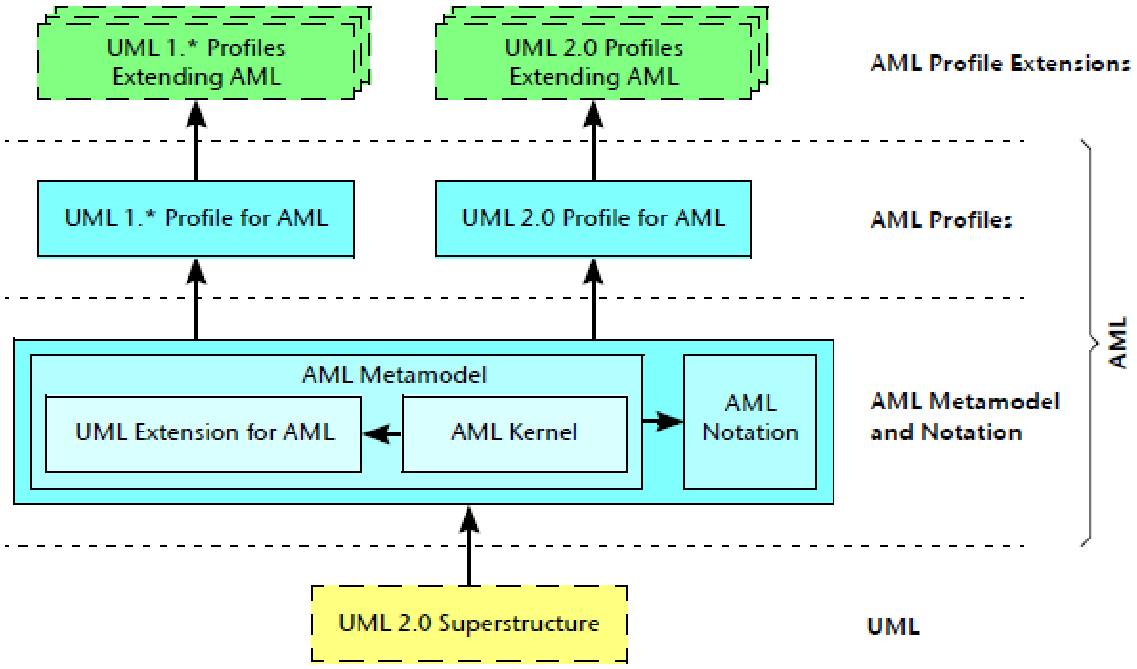

Figure 2: Levels of AML Specification (from [16])

The AML Profiles packages enable the implementation AML within CASE tools, which are based on UML 1.* and UML 2.0. As it is described in [18], users are free to define their own language extensions to customize AML for their needs. These extensions can be defined also as UML 1.* or UML 2.0 profiles, commonly referred as AML Profile Extensions. Such an extension is core part of this thesis.

2.3 Elements of AML

In this section we will give an overview of AML packages and elements that are closely related to our work.

2.3.1 Mental Package

The AML specification [18] describes this package as: "The Mental package defines metaclasses which can be used to

- support analysis of complex problems/systems, particulary by:
  - modeling intentionality in use case models,
  - goal-based requirements modeling,
  - problem decomposition, etc.
- model mental attitudes of autonomous entities, which represents their informational, motivational and deliberative states."

The Mental package can be divided to more sub-packages, as shown on Figure 3.

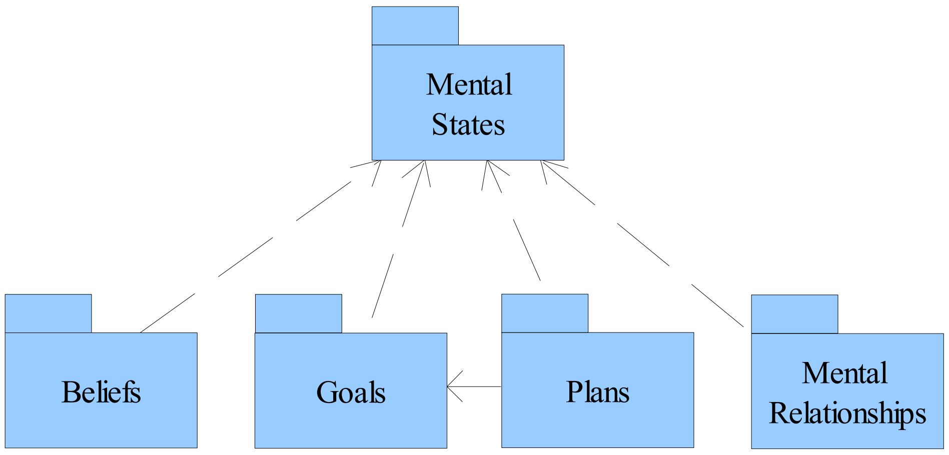

Figure 3: The structure of mental package

Metal States package defines fundamental metaclasses, which are used to specify metaclasses in other sub-packages. Beliefs, Goals, Plans sub-packages as their name denotes, define elements for capturing corresponding terms, as can be seen this structure corresponds to BDI paradigm (see section 3.1). The Mental Relationship sub-package defines relations between mental elements (more precisely between Mental States, see [18]) to support reasoning processes.

Belief

Stereotype: <<belief>>

Belief is specialized MentalClass used to model a state of affairs, proposition, or other information relevant to the system and its mental model. The specification of information is expressed by the owned constraint. It is possible to specify attributes and/or operations for a Belief, to represent its parameters and functions, which can both be used in the owned constraint as static or computed values.

Goal

Stereotype: Goal is an abstract element, therefore has no general notation.

Goal is an abstract element, introduced to define the common features of all its subclasses that are used to model concrete types of goals. It defines common semantics of a AML Goals, that can be characterized as conditions or states of affairs, with which the main concern is their achievement or maintenance. The Goals can thus be used to represent objectives, needs, motivations, desires, etc.

DecidableGoal

Stereotype: <<dgoal>>

DecidableGoal is used to model goals for which there are clear-cut criteria according to which the goal-holder can decide whether the DecidableGoal (particularly its postCondition) has been achieved or not. The DecidableGoal rectangle can contain special compartments <<commit>>,<<pre>>, <<inv>>, <<cancel>>, and <<post>> for the contained MentalConstraints, these represent predefined MentalConstraintKinds (see definition of MentalConstraintKind below) . These compartments may be omitted and can be specified in any order.

UndecidableGoal

Stereotype: <<ugoal>>

UndecidableGoal is a specialized concrete Goal used to model goals for which there are no clear-cut criteria according to which the goal-holder can decide whether the postCondition of the UndecidableGoal is achieved or not.

Plan

Stereotype: <<plan>>

Plan is used to model capabilities (of MentalSemiEntityTypes of AML) which represents either:

- predefined plans, i.e. kinds of activities a mental semi-entity's reasoning mechanism can manipulate in order to achieve Goals, or
- fragments of behavior from which the plans can be composed (also called plan fragments).

In addition to UML Activity, Plan allows the specification of commit condition, cancel condition, and invariant (for details see MentalConstraintKind), which can be used by reasoning mechanisms.

For modeling the applicability of Plans, in relation to given Goals, Beliefs and other Plans, the Contribution relationship is used.

Contribution

Stereotype: <<contributes>>

Contribution is a specialized MentalRelationship and DirectedRelationship (from UML) used to model logical relationships between MentalStates and their MentalConstraints. The manner in which the contributor of the Contribution relationship influences its beneficiary is specified by values of meta-attributes of the particular Contribution.

MentalAssociation

Stereotype: MentalAssociation is depicted as a binary UML Association with the stereotype <<mental>>

MentalAssociation is introduced to enable modeling of MentalProperties in the form of association ends. It is used to specify that mental semi-entities have control over Goal and Belief instances.

MentalConstraintKind

MentalConstraintKind is an enumeration which specifies kinds of MentalConstraints, as well as kinds of constraints specified for contributor and beneficiary in the Contribution relationship.

Value	Keyword	Semantics
commitCondition	commit	An assertion identifying the situation under which an autonomous entity commits to the particular ConstrainedMentalClass (if also the precondition holds).
preCondition	pre	The condition that must hold before the ConstrainedMentalClass can become effective (i.e. a goal can be committed to or a plan can be executed).
commitPreCondition	commpre	AND-ed combination of commitCondition and preCondition. Used only within Contribution.
invariant	inv	The condition that holds during the period the ConstrainedMentalClass remains effective.
cancelCondition	cancel	An assertion identifying the situation under which an autonomous entity cancels attempting to accomplish the ConstrainedMentalClass.
postCondition	post	The condition that holds after the Constrained-MentalClass has been accomplished (i.e. a goal has been achieved or a plan has been executed).

2.3.2 Architecture Package

The Architecture package defines the metaclasses used to model architectural aspects of multi-agent systems. These aspects are captured in more sub-packages like Agents, Resources, Environments etc.. However just few of these concepts are related to our work.

AgentType

Stereotype: <<agent>>

AgentType is a specialized AutonomousEntityType used to model a type of agents, i.e. self contained entities that are capable of autonomous behavior within their environment.

AgentType can use all types of relationships allowed for UML Class, e.g. associations, generalizations, dependencies, etc., with their standard semantics, as well as inherited AML-specific relationships.

EntityRoleType

Stereotype: <<entity role>>

EntityRoleType is used to represent a coherent set of features, behaviors, participation in interactions, it is introduced to model roles in multi-agent systems.

2.3.3 Behaviors Package

The Behaviors package defines the AML metaclasses used to model behavioral aspects of multi-agent systems, as behavior decomposition, mobility, communicative interactions.

CommunicationMessagePayload

Stereotype: <<cm payload>>

CommunicationMessagePayload is a specialized Class (from UML) used to model the type of objects transmitted in the form of CommunicationMessages.

3 Overview of Jadex

In this section we first provide a short description of the BDI software model, that served as a motivation for architecture of Jadex. Then we will introduce a detailed description of Jadex system and its programming model. We will omit details which are not relevant to the scope of this thesis. Some programming model elements will be described in more detail, because are necessary for understanding the system's relation to AML. For more detailed documentation refer to Jadex homepage [2]. Most parts of this section are based on Jadex User Guide [1] and Jadex Tutorial [10].

3.1 The BDI Software Model

The Belief-Desire-Intention model is a software architecture for development of intelligent software agents. In these architecture, the internal design and the process of choosing a course of action, is driven by mental attitudes. The advantage of this approach, thus using mental attitudes for design or realization, is that it provides a more human-like abstraction, therefore simplifies the understanding of system. This concept was first introduced by Michael Bratman [13].

The idealized components of a BDI agent are:

- Beliefs – represent the agent's thoughts about the world, or informational state of an agent. Term belief, instead of knowledge, denotes that agent's beliefs are not necessary true.
- Desires – represent the objectives, or motivations of an agent. A special type of desire is Goal, which assumes that the active desires of an agent are consistent.
- Intentions – represent the deliberative state of an agent, thus desires that agent committed to do. In software systems these intentions are kind of plans, thus set of actions, which might lead to accomplishing one or more of its intentions. Plans could be separated to more sub-plans.
- Events – are kind of triggers that has impact on agents beliefs, desires, intentions. Event can be external, received by sensors, or internal generated by reasoning system.

There can be described a general reasoning system of an agent, that can perform complex tasks in dynamic environments. A very simplified version can be denoted as an infinite loop consisting of 3 steps: sense, select, act – where an agent observes its environment, it selects and executes an action.

This BDI model is closely related to both systems that are in focus of our observation.

3.2 The Jadex Reasoning Engine

Jadex is an Agent oriented reasoning engine based on BDI paradigm, described above. It can be used together with kinds of agent middleware, that provides basic agent services, such as communication infrastructure or management facilities.

Rational agents in Jadex have explicit representation of their environment and objectives that they are trying to achieve. In this case rationality means that agent always performs the most promising step to achieve its objectives. In Jadex belief, goals and plans are first class objects, that can be accessed inside an Agent.

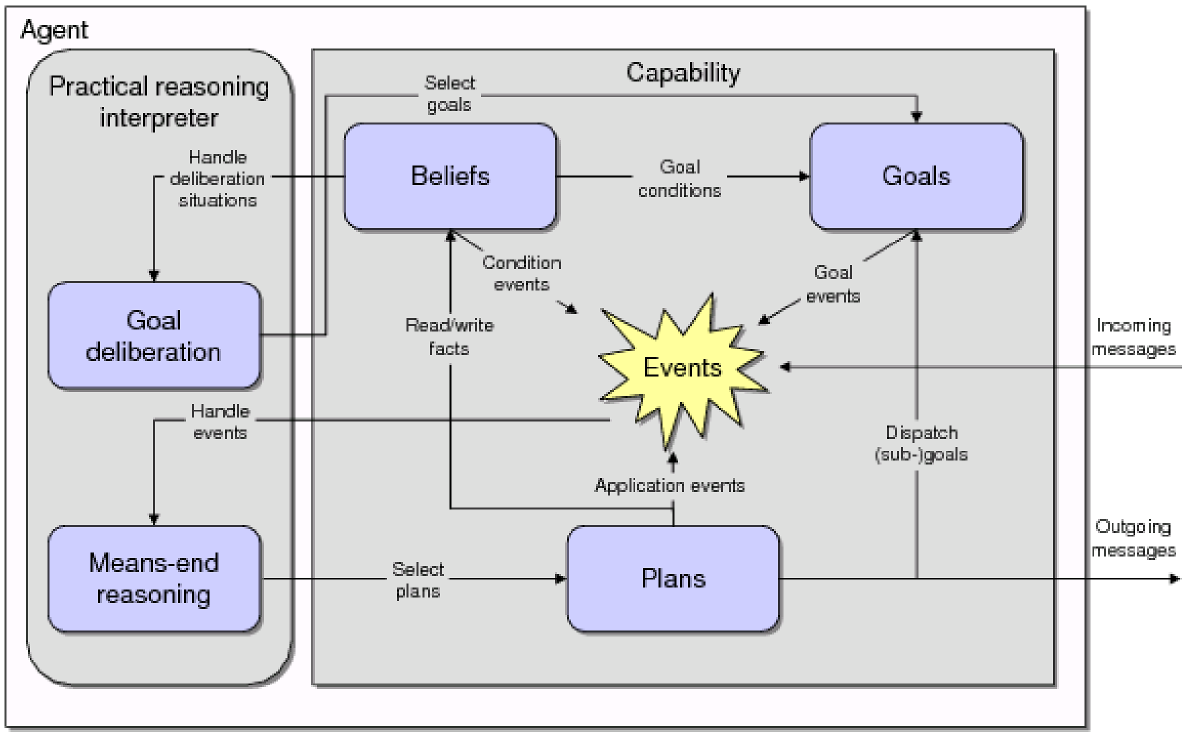

Figure 4: Jadex Abstract Architecture (from [1])

The reasoning in Jadex can be described as a process consisting of two interleaved components. On one hand, the agent reacts to incoming messages, internal events and goals, by selecting and executing plans. On the other hand, the agent continuously deliberates about its current goals, to decide about consistent subset, which should be pursued.

Jadex specific Beliefs are arbitrary java objects which can be stored in “beliefbase”. The “beliefbase” stores these objects as believed facts, it is also an access point for agent to its data. The belief representation is very simple, the “beliefbase” contains strings which are identifiers to a specific Belief or its value.

Goals are kind of motivations which inspires the Agents behavior. They are central components of Jadex, which follow the concept that goals are the actual desires of the agents. Therefore the agent will be directly engage into suitable actions until the goal is reached, or is unreachable, or not desired any more. Jadex does not assume that all adopted goals are need to be consistent to each other, it provides a life-cycle management for goals, which defines three states for goals: option, active, and suspended (see Figure 5). The system also provides an application specific goal deliberation mechanism, which is responsible for managing state transitions of all adopted goals. Additionally a goals state depends on context determined by agent's beliefs. When a context is invalid the depending goal is suspended until it is valid again.

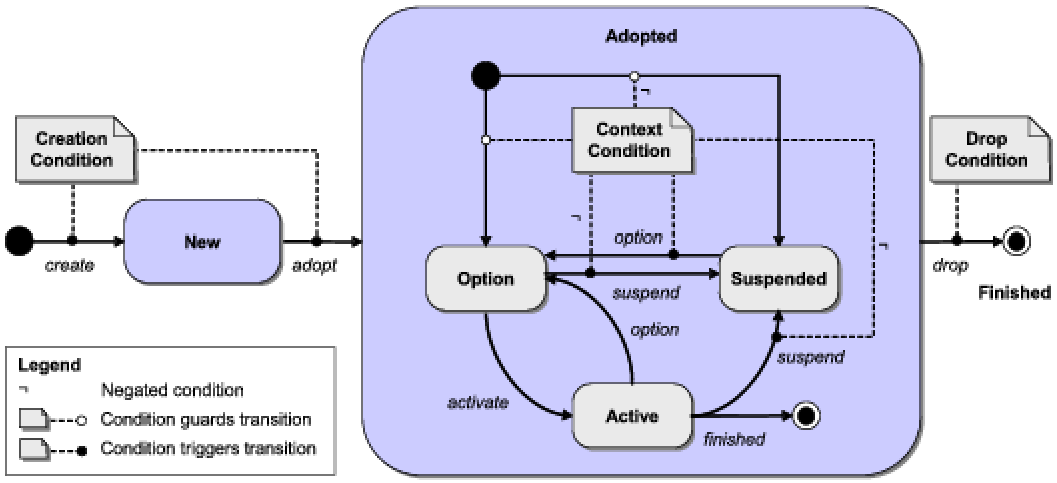

Figure 5: Goal Life-cycle (from [1])

The system defines four basic types of goals:

- Perform goal – denotes that something needs to be done, but there is no explicit desirable result defined.
- Achieve goal – describes a target state to be reached, but it not specifies how to reach it. Therefore agent may try different alternatives.
- Query goal – indicates that some information needs to be acquired. If the information is not available, plans are executed to reach them.
- Maintain goal – describes some state that should be kept after once it is achieved. It is the most abstract goal in the system. It abstracts from actions needs to be done to achieve the goal, and decouples the creation and adoption of the goal from time-point when it is executed.

There is also a semantically distinct type of goal, called Meta goal, which is used for meta level reasoning. For example if there are multiple plans matching for a goal then corresponding metal level plans are executed to achieve the Meta goal, thus to make a selection between them.

As we mentioned before, one aspect of rational behavior is that agent can pursue multiple goals in parallel. The system provides an architectural framework for deciding how goals interact and how an agent can autonomously decide which goal to pursue. This process is called goal deliberation. Jadex supports a goal deliberation strategy called easy deliberation, which is a simple and elegant way to allow developers to specify relationships between goals in intuitive manner. It is based on goal cardinality, which restrict the number of goals of a given type that can be active at once, and goal inhibitions, which prohibit certain other goal to be pursued in parallel.

Plans can be interpreted as recipes for achieving goals. Plans consist of two parts, the header and the body. Header is a kind of definition of a plan, where some attributes and conditions of execution are specified. The body itself is implemented in java programming language, giving the system high flexibility. Plans are instantiated at runtime. Activation triggers in header are used to specify when a plan should be instantiated. In addition some initial plans can be executed when the agent is born. During the execution, plans have also possibility not just execute arbitrary java code, but also dispatch sub-goals or to respond to events.

3.3 The Programming Model

Development of agents consists of creation of two types of files: plan implementations in Java programming language, and an XML file called the Agent Definition File (ADF). In ADF an agent itself is described. If we start an agent first this file is loaded and corresponding belief, goals, and plans are created as specified.

3.3.1 The Agent Definition File

The ADF can be interpreted as a specification of an agent. The general structure of the file is shown on Figure 6. The name of the agent type is defined in name attribute of the root (<agent>) tag. This name is also a prefix for the name of the ADF XML file and should be followed with ".agent.xml". An other important attribute is `package` which should correspond to location of the file, also it is used as "classpath" where system searches for the required classes.

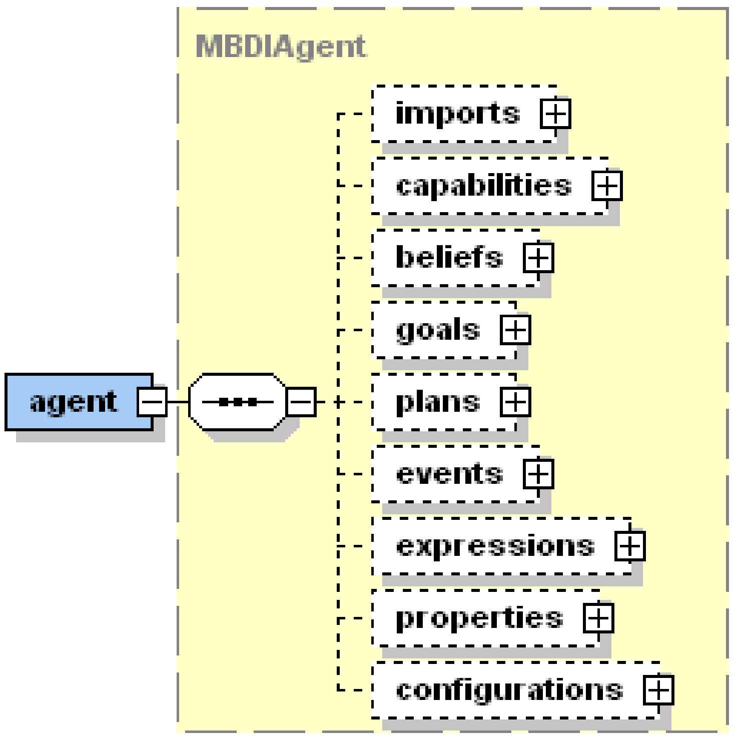

Figure 6: The structure of Agent Definition Files (from [1])

Imports

In imports section classes and packages are defined, which can be used by Java expressions in ADF. In addition paths to non-Java artifacts like agent xml or capabilities are specified this way.

Capabilities

In Jadex Capabilities are used to modularize common agent behaviors. Practically they are agents without own reasoning process – also their definition is almost same, just the `agent` tag is replaced by `capability` tag. Every agent has at least one capability, its own beliefs, goals and plans. Also an agent can be seen as a collection of capabilities and an additional reasoning process between them. Agents and Capabilities can have an arbitrary number of sub-capabilities, defined under `capabilities` tag. To use a capability, we must specify its definition file, and a name through which it can be referenced.

Beliefs

```
<belief name="simpbeliefbase" class="int">
    <fact>222</fact>
</belief>
<beliefset name="simpBeliefList" class="String">
    <facts>getStrings()</facts>
</beliefset>
```

Figure 7: Example of a simple Belief

Beliefs are facts known by an agent, which can be modified inside a plan. To define a single or multivalued belief the corresponding `belief` or `beliefset` tag is used. Developer has to specify class and name of a belief with identical attributes. The default or initial beliefs are supplied in enclosed `fact` tags, in case of multivalued beliefs the list of these elements is enclosed in `facts` tag.

Goals

In Jadex the four goal kinds are strongly typed, all of them is expressed in ADF with corresponding tag: `maintaingoal`, `achievegoal`, `performgoal`, `querygoal`. However all of them are semantically different, but they share some common attributes. All of the attributes can be identified by its name. All parameters of the goal have to be declared in the XML file. These declarations of parameters resemble the specification of beliefs, thus the single and multivalued parameters are distinguished. Also parameter values can be defined using expressions. The system distinguishes `in`, `out`, and `inout` parameters specified by direction attribute, depending on when a parameter is used, or when it is set. The `unique` tag for a goal denotes that only one instance of a goal can be adopted at same time.

To describe situations when a goal has to be automatically instantiated, the `creationcondition` is used. To denote situations when the goals needs to be suspended or dropped the `contextcondition` and `dropcondition` elements are used. Specific goals has also their own specific types of elements and attributes, for example Maintain goal has its special `maintaincondition` element. For detailed specification please refer to Jadex documentation [1].

```
<achievegoal name="achievecleanup" retry="true" exclude="when_failed">
    <parameter name="waste_location" class="Location">
        <value>$beliefbase.known_waste_location</value>
    </parameter>
    <creationcondition>
         $beliefbase.getBeliefSet("known_waste_locations").size()>0
    </creationcondition>
    <contextcondition>
        $beliefbase.daytime
    </contextcondition>
    <dropcondition>
        !$beliefbase.carrieswaste
    </dropcondition>
    <deliberation cardinality="1">
        <inhibits ref="performlookforwaste"/>
        <inhibits ref="achievecleanup"/>
    </deliberation>
</achievegoal>
```

Figure 8: Example of a goal definition in ADF

Goal deliberation

The goal deliberation settings are included in the goal specification, using the `deliberation` tag (see Figure 8). The cardinality is specified as an integer value with cardinality attribute, by default this value is unlimited. Inhibition is denoted using ref attribute within `inhibits` tag. This reference specifies the goal to inhibit. Additionally an condition can be specified as content of inhibit element. The inhibition only takes effect when this condition is true.

Plans

Inside `plans` element an arbitrary number of plan headers can be defined using `plan` elements. Each plan can have several attributes. The `name` attribute is mandatory. The `priority` attribute is also important to define preferences between plans. For each plan the `body` element has to be provided, which specifies the implementing java class. Within this element a java expression can be defined that creates the instance, or simply we can specify the name of the class that implements the plan, using the `class` attribute.

```
<plan name="moveto">
    <body class="MoveToLocationPlan"/>
    <trigger>
        <goal ref="achievemoveto"/>
    </trigger>
    <contextcondition>
        $beliefbase.chargestate &gt; 0
    </contextcondition>
</plan>
```

Figure 9: Example of a plan definition in ADF

To indicate when a plan is applicable or shall be created, the `trigger` element can be used. The most common situation for triggering a plan is reacting to a goal actualization, but Jadex introduces a more general model, thus with sub-tags of the `trigger` element we can specify also internal-, message events for which a plan is applicable. In addition it is also possible to define data driven execution by using `condition` tag. To generalize the concept also precondition and context condition can be introduced with corresponding tags.

Events

Agent have the property to react to different kind of events. Jadex differentiates two kind of events. Internal events are kind of one-way communication of occurrences inside of agent. A typical use case of this event is a GUI update. The other type, which is more interesting for us, are message events, which are used for communication between autonomous agents. All messages has to be specified in ADF, and as goals has arbitrary number of parameters. Messages has several properties and flags, which follow the FIPA-ACL [49] standard (Jadex is not restricted only on this type of messages, however only these are available in current release). The templates of messages are defined in `events` section, using `messageevent` element. All the FIPA-ACL parameters are created automatically, a detailed description about them you can see in [1] or in FIPA-ACL specification. The `direction` attribute is used to define if an agent wants to send, receive or both, the given event. The `content` parameter is used to define the data transmitted by the message.

The actual sending and receiving a message is realized within plans, through corresponding methods.

```
<messageevent type="fipa" name="SampleMessage"  direction="send"
       posttoall="true" randomselection="true">
    <parameter name="performative" class="String"
              direction="fixed">
        <value>SFipa.INFORM</value>
    </parameter>
    <parameter name="content" class="TransferData">
        <value>new TransferData()</value>
    </parameter>
</messageevent>
```

Figure 10: Example of a message event definition

Configurations

Within `configurations` element in ADF, "initial" and/or "end" state of an agent type can be defined. Initial instances of elements like goal or plan can be declared, thus are created when the agent is started. On the other hand "end" elements can be specified, which are instantiated when the agent is going to be terminated. Instances of elements always have to refer some already declared element using `ref` attribute. Arbitrary number of configurations can be defined for an agent or capability; each of them must have a name for identification purpose. When starting an agent an arbitrary configuration can be choose, also a default configuration can defined by attribute with identical name. Configurations allow us to specify elements like capabilities, beliefs, goals, plans and events.

Within `capabilities` element the initial configuration of a referenced capability can be set. A capability can also have more configurations, when it is included, by `initialcapability` attribute can be specified which one to use. In the `beliefs` section initial beliefs of belief sets can be altered. Thus we can newly introduce the facts in referenced beliefs using `initialbelief` and `initialbeliefset` elements. Within `goals` and `plans` elements, as we told before, "initial" and "end" goals and plans can be

```
<configuration name="benchmark">
    <capabilities>
        <initialcapability ref="SampCapability" configuration="ConfigA"/>
    </capabilities>
    <beliefs>
        <initialbelief ref="quiet">
            <fact>true</fact>
        </initialbelief>
    </beliefs>
    <plans>
        <endplan ref="benchmark">
            <parameter ref="goals">
                <value>500</value>
            </parameter>
        </endplan>
    </plans>
</configuration>
```

Figure 11: Example of a configuration in ADF

defined, furthermore values of their parameter can be redefined. In `events` section "initial" and "end" events can be specified, which are instantiated when the agent is born.

3.3.2 Plan Implementation

As we told before Plans are implemented using Java programming language. There are two types of plans in the system, standard and mobile plans. When we implement one of them, we extend `jadex.runtime.Plan` in case of standard plans, and `jadex.runtime.Mobile` plan in case of mobile plans. The code of standard plan is placed in `body()` in the other case to `action(IEvent)` method. Jadex provides a library that allows us to access the object described in ADF. For more detailed description see [1].

4 AML to Jadex Mapping

This section is the theoretical part of our contribution. Here we specify a mapping between AML and Jadex platform. In the focus of our discussion is the relation between Mental package of AML and Agent Definition Files (ADF). As can be seen from overview of Jadex and AML Mental package the semantics of their elements are close to each other, since both of them follow the BDI paradigm. Wee will see that the mapping between these systems allow us to generate detailed source code.

Jadex plans, which are other components used to develop agents within the system, are implemented in Java programming languages, using libraries provided in Jadex. Detailed code generation of these files is beyond the scope of this thesis, since it goes behind scope of AML Mental package. On the other hand Jadex, has only a very general specification of these files, thus no additional strict implementation constraints are defined, therefore detailed generation of such files is very close to problem of code generation from UML models to Java language. We will cover this topic in section 7.2.

As we mentioned before our goal is to generate highly detailed Agent Definition Files. To achieve this goal we additionally specify an extension of AML profile or variant of AML. We introduce some extensions in form of tagged values for existing elements. In addition we define stereotypes that capture elements of ADF, that cannot be properly described using elements of AML either UML, thus there are no constructs with corresponding semantics, but are fundamental parts of Jadex agents.

The specification of mapping is structured as follows:

- Mapping – mappings of AML elements to corresponding Jadex source code fragments.
- Extensions – extensions for AML profile with Jadex platform related platform specific elements, and their mapping specifications to source code fragments.
- Constraints – describes rules or constraints that have to be followed to achieve correct code generation.
- Rationale – additional explanations, discussions and justification for mapping.

4.1 Mental Aspects

Mapping of elements from Mental package.

4.1.1 DecidableGoal

Mapping

- DecidableGoal is mapped to Achieve goal in ADF (in case the `goaltype` tagged value is not specified).

- MentalConstraints of DecidableGoal are mapped as defined in section 4.1.6.

Extensions

- To denote that the DecidableGoal is kind of Jadex specific goal we use tagged value with tag name `goaltype`, its value is one of five types of goals with lowercase letters: `achievegoal`, `performgoal`, `querygoal`, `maintaingoal`, `metagoal`. Depending on this tagged value the corresponding XML element is generated into ADF.
- In addition the common goal attributes: `retry`, `retrydelay`, `exclude`, `posttoall`, `randomselection`, `recur`, `recurdelay` are denoted as tagged values, with identical name. These are generated as attributes of corresponding `goal` tag, their value is translated without modification. For detailed description of these attributes see [1].
- The `unique` boolean tagged value, is introduced to denote the goal is unique. It is mapped to `unique` empty sub-element of a `goal` element, in case its value is true.
- The `cardinality` tagged value is introduced to denote cardinality of a goal; it is mapped to `cardinality` attribute of `deliberation` sub-element of Jadex specific goal elements.
- The `exported` tagged value is introduced, to denote that the goal is exported within a capability. It is mapped to `exported` attribute of the `goal` tag, with identical name and value.
- The `abstract` boolean tagged value is introduced, to denote that a goal is abstract within a capability. In case its value is true, it is mapped to an empty `abstract` sub-element of the goal. In this case it is not the standard goal tag that is generated, but its reference name within a capability, for example `achievegoalref` instead of `achievegoal`.
- The Attributes with `Parameter` stereotype are mapped as described in section 4.4.5.

Constraints

A goal is a reusable standalone element which can be associated with more agents or capabilities. To denote which AgentType or EntityRoleType has the specified DecidableGoal, a MentalAssociation is used. For more details see the mapping of MentalAssociation in section 4.1.4

Tagged values `exported` and `abstract` are used for code generation only in case that DecidableGoal is associated with EntityRoleType.

Rationale

AML has two specialized concrete goals, DecidableGoal and UndecidableGoal. DecidableGoals are used to model goals for which the goal-holder can decide whether it has been achieved or not; in contrary with UndecidableGoals which cannot be decided. The semantic of DecidableGoal element corresponds to Jadex's general goal concept, thus in Jadex all goals are decidable in sense that all their specific conditions can be evaluated at any time.

Achieve goal is selected as default goal to be generated, since it is the most intuitive type of goal in Jadex.

There is no corresponding notion to a goal's degree in Jadex, what in AML denotes kind of reliability or confidence; we can interpret goals in Jadex as goals with maximal degree.

4.1.2 Plan

Mapping

- Plan is directly mapped to Jadex `plan` element in ADF. The name of the plan is mapped to `name` attribute of `plan` element in ADF.
- The `degree` attribute of AML Plan is mapped to `priority` attribute of `plan` tag.
- The skeleton of the class implementing the Jadex plan is generated from AML Plan. In case it is not specified explicitly by an extension, skeleton of standard plan is generated. The name of the plan is mapped to the name of the class which implements the plan (with upper-case first letter according to Java naming conventions).
- MentalConstraints of Plan are mapped as defined in section 4.1.6

Extensions

- To denote the name of the class which implements the plan, we introduce `bodyclass` tagged value, which is used also to generate the `class` attribute of `body` element in ADF (but only in case the `body` tagged value is not presented, see the following extension).
- In case the more specific instantiation of the implementing class is required, the `body` tagged value is introduced, its value is mapped to the value of the `body` element.
- To differentiate between two kinds of Jadex plans the `plantype` tagged value is introduced. In case its value is `mobile` then mobile plan, in case its value is `standard` then standard plan is generated. The type of the plan is denoted also in ADF, by the `type` attribute of `body` tag; this attribute is also generated from this tagged value. In case this tagged value is not specified, skeleton of a standard plan is generated.
- The `exported` tagged values are introduced with identical usage as defined in extension section of DecidableGoal.
- The Attributes are mapped using Parameter stereotype, as described in section 4.4.5.

Constraints

Plan is a reusable standalone element which can be associated with more agents or capabilities. To denote which AgentType or EntityRoleType has the specified Plan the MentalAssociation is used.

AML does not specify either the syntax or semantics of degree value, however in Jadex the priority attribute is an arbitrary integer value, we assume that the value is provided in form according to Jadex specification.

Rationale

AML Plan semantically corresponds to Jadex Plan.

In Jadex the class which implements a plan can be specified by introducing its name or, in a more general way, by introducing a Java code snippet which instantiates it, for example a constructor call. Our intent was to cover both cases with `body` and `bodyclass` tagged values.

4.1.3 Belief

Mapping

The constraint of Belief is mapped to a Jadex `condition` element:

- name of the Belief is mapped to `name` attribute of the condition,
- value of the constraint is mapped as value of the `condition` element.

The Attributes of Belief with `beliefbase` stereotype are mapped as defined in section 4.4.4.

Constraints

There must be a MentalAssociation between AgentType (in case of capability between EntityRoleType) and Belief.

Rationale

Semantics of AML Belief differ from semantics of beliefs in Jadex, in the sense that an AML Belief is rather a logical expression which can be evaluated; on the other side Jadex beliefs are data objects holding information state of an agent. Therefore semantics of AML Belief correspond more to Jadex conditions, which are monitored boolean expressions as defined in [1]. However, in both cases, these conditions may be dependent on attributes of belief, therefore Jadex beliefs are also generated from AML Belief, thus from their Attributes.

There is not a corresponding notion to Belief degree in Jadex, what in AML denotes kind of reliability or confidence; we can interpret beliefs in Jadex as beliefs with maximal degree.

4.1.4 Mental Association

Mapping

The default constraints and mapping of MentalAssociation is denoted in constraints section of DecidableGoal, Plan and Belief mapping. Here we define additional extensions for certain situations.

Extensions

The following extensions are used to capture the case when a plan, goal, or a belief is exported from a capability and included to an agent. The MentalAssociation between AgentType and above mentioned MentalClasses may have the following tagged values:

- `fromcapability` boolean tagged value (if it is true) denotes that the corresponding plan, goal or belief is included from a capability, or it is assigned to a corresponding abstract element of a capability.
- `capabilitylocalname` tagged value is introduced to denote the name of the capability from which the specified element is included.

Rationale

AML defines MentalAssociation as a specialized association (from UML) between a MentalSemiEntityType, in this case the EntityRoleType, and between a MentalClass, in this case a DecidableGoal, Plan or Belief. This is a typical usage of this connector, see examples in [18].

Extensions are defined in order to facilitate code generation in situation:

- Where a concrete goal or belief (the imported element) is imported from a capability that exports it. In this case the element is connected using MentalAssociations both with the EntityRoleType (the capability where the element is defined), and with the AgentType (or EntityRoleType; where it is imported).
- Where a concrete goal or belief is assigned to a corresponding abstract element of a capability. For further description see section 4.4.6.

The `capabilitylocalname` tagged value is introduced since there could be a situation when an imported element is associated with more capabilities that are included to the agent. By defining this tagged value we avoid the possible ambiguities.

To clarify these extension see example in section 5.3.

4.1.5 Contribution

Mapping

Contribution is mapped into:

1. Trigger of a Goal, in case it is between a DecidableGoal and a Plan. Thus `goal` sub-element of `trigger` element of a plan. The name of the DecidableGoal is mapped

to `ref` attribute of `goal` tag.

2. Condition of a trigger, in case it is between a Belief and a Plan. Thus `condition` sub-element of `trigger` element of a plan. The `constraint` of Belief is mapped to value of `condition` element.
3. Condition of a goal, in case it is between a Belief and a DecidableGoal. The type of the condition element is determined from the BeneficiaryConstraintKind according to section 4.1.6.
4. Inhibition, in case it is between two DecidableGoals. Thus to `inhibition` sub-element of `deliberation` element of a goal. This is a negative Contribution, thus contributor inhibits the beneficiary. The `inhibition` element is generated to goal definition of beneficiary.
5. Trigger of meta goal, in case it is between a goal witch is not a meta goal, and between a meta goal. Thus `goal` sub-element of `trigger` element of meta goal) The name of the non meta goal is mapped to `ref` attribute of goal tag.

Extensions

For case 4, we introduce `invariant` tagged value to express condition of `inhibition;` in addition we introduce `inhibit` tagged value to express the identical attribute of `inhibition` tag, with possible values of `when_active` or `when_in_process`.

Constraints

In case 3, combination of more sufficient and necessary Contributions is allowed, that is translated as defined in [18]. The necessary Contributions on same MentalConstraintKinds are logically AND-ed, and sufficient MentalConstraintKinds are logically OR-ed. It is assumed that only Contributions with identical ContributionKinds affect on one MentalConstraintKind of a DecidableGoal.

Rationale

Contribution has generic semantics in AML. It is used to model various kinds of mental or logical relations. The manner in which the contributor of the Contribution relationship influences its beneficiary is specified by values of meta-attributes of the particular contribution. Although meta-attributes does not influence the generated code (except case 3), in the following we specify them according to mappings, thus we clarify whether a certain situation is modeled properly, to be semantically correct:

1. In general sense, from the point of view of Jadex, this is a sufficient contribution, where the ContributorConstraintKind is CommitPreCondition, and the BeneficiaryConstraintKind attribute of contribution is CommitCondition. This means that the CommitCondition stands for the trigger of a plan in Jadex. Thus in Jadex terminology, if the goal is committed (pre- and commit conditions hold) it is sufficient for the execution of the plan. This also corresponds to situation when there are more triggers defined for the plan; in this case the triggers are logically OR-ed in Jadex.

   The value of the `degree` attribute depends on the actual situation. Lets assume that there are more plans associated with a goal, in this case a meta level reasoning could be executed for selecting the appropriate plan, therefore value of the degree depends on the implementation of an actual reasoning.

2. Analogous to the first case, only the ContributorsConstraintKind is unspecified. According to Contributions definition in this case the Belief's constraint is considered to contribute.
3. The ContributionKind is optional. However if only one Belief contributes to a condition, then from aspect of Jadex it is a logical equivalence.

   The ContributorsConstraintKind is unspecified. According to the definition of Contribution, in this case the Belief's constraint is considered to contribute. The BeneficiaryConstraintKind is optional. According to these rules the `degree` of the contribution, if there is only one Belief that contributes to the condition, is maximal. In other cases it depends on the situation.
4. In this case due to the easy deliberation mechanism implemented by Jadex, ContributionKind is sufficient, thus there is no additional constraint that must hold to inhibit the contributor. The `degree` is minimal, in other words this is negative contribution. Due to the nature of this relation between Jadex goals, the ContributorsConstraintKind and the BeneficiaryConstraintKind is not defined.
5. This is a sufficient Contribution, thus in this case dispatching such goal is sufficient for execution of Meta Goal. In this sense the ContributorConstraintKind is CommitPreCondition and the BeneficiaryConstraintKind is CommitCondition of Contribution. The `degree` of Contribution is naturally maximal.

4.1.6 MentalConstraintKind

Mapping

The following table defines how the MentalConstraintKind attributes are mapped to Jadex elements.

MentalConstraintKind / Jadex Elements	**Achieve goal**	**Perform goal**	**Query goal**	**Maintain goal**	**Meta goal**	**Plan**
preCondition						Pre-condition
commitCondition	Creation condition	Creation condition	Creation condition	Creation condition		Condition of Trigger
invariant	Context condition	Context condition	Context condition	Maintain condition	Context condition	Context condition
cancelCondition	Failure condition	Drop condition	Drop condition	Drop condition	Failure condition	
postCondition	Target condition			Target condition	Target condition	

Rational

The MentalConstraintKind is defined in [17] as an "enumeration which specifies kind of MentalConstraints, as well as kinds of constraints specified for contributor and beneficiary in Contribution relationship". Literals of this enumeration have their semantically similar or corresponding representatives in Jadex, as defined by the table above. Some of the Jadex goals, like Maintain goal and Achieve goal, define their special condition, additionally to common conditions. These special conditions has priority to common conditions, in case more of Jadex conditions could be represent a MentalConstraintKind, for example the Invariant MentalContraint could be represented by Context condition but also by Maintain condition in Achieve goal. In this case we gave priority to Maintain condition because its denotes a specific property of this goal.

4.2 Architectures

Mapping of elements from Architectures package.

4.2.1 AgentType

Mapping

- AgentType is mapped to an Agent Description File. The file name will have the form: the name of agent followed by ".agen.xml" suffix. The root `agent` element is generated to ADF with static parts as schema definition and schema location.
- Name of AgentType is mapped to `name` attribute of `agent` tag.
- The Namespace of the AgentType is mapped to `package` attribute of `agent` tag.

Rationale

In AML AgentTypes are used to model types of agent, thus self-containing entities that are capable of autonomous behavior within their environment, what corresponds to a Jadex agent.

4.2.2 EntityRoleType

Mapping

- EntityRoleType is mapped to Capability, in an Agent Description File. The file name will have the form: name of the capability followed by ".capability.xml" suffix. The root `capability` element is generated to ADF with static parts like schema definition and schema location.
- The name of the EntityRoleType is mapped to `name` attribute of `capability` tag.
- The Namespace of the EntityRoleType is mapped to package attribute of capability tag.

Constraints

There must be a PlayAssociation between an AgentType and EntityRoleType to denote the agent plays the role, thus in Jadex terms it includes the Capability.

Rationale

EntityRoleType in AML is used to represent a coherent set of features behaviors, participation in interaction, or services offered or required by behavioral entities. This definition corresponds to Capability in Jadex, what is basically an agent but without an own reasoning process, thus a collection of features as goals, plans, etc..

4.2.3 PlayAssociation

Mapping

- If PlayAssociation is between AgentType and EntityRoleType or between two EntityRoleTypes, then a capability include is generated to the ADF.
- The name of the EntityRoleType is mapped to `name` attribute of capability tag.
- The value of `file` attribute of `capability` tag is created from concatenation of namespace of destination EntityRoleType and its name is separated with a dot.

Constraints

In case PlayAssociation is between two EntityRoleTypes it must be a directed association, to denote only the source capability includes the target capability.

Rationale

PlayAssociation as explained in AML, is introduced to model the possibility of playing entity roles by behavioral entities. Therefore, this case is a typical usage of PlayAssociation.

4.3 Behaviors

Mapping of Elements from Behaviors package.

4.3.1 CommunicationMessagePayload

Mapping

- CommunicationMessagePayload is mapped to a Message Event (`meesageevent` sub-element of `events` element in ADF), its name is mapped to `name` attribute of `messageevent` tag.
- The `performative` tagged value is mapped to a special parameter of Message Event. The value of `name` attribute will be `performative`, value of `type`

attribute will be `fipa`, and value of direction attribute will be `fixed`. The value of `performative` tagged value is mapped to the value of `value` sub-element of the parameter.

- In case no attribute with `parameter` stereotype and name `content` is introduced and the value of `abstract` tagged value is not true, attributes that have no Parameter stereotype are mapped to fields of a Java class that represents the data transferred by the Message Event (encapsulation methods for fields also generated, the name of the CommunicationMessagePayload is mapped to name of the Java class). If there are no such Attributes no Java class is generated. The corresponding `content` parameter is generated to ADF, that defines that the content of the Message Event will be the generated class.

Extensions

- The Attributes with `parameter` stereotype are mapped as defined in section 4.4.5. List of valid parameters for Message Events can be found in Jadex User Guide [1].
- The `direction` tagged value is introduced with possible values of `send`, `receive` and `send_receive`, to denote the identical attribute of `messageevent` tag.
- The `type` tagged value is introduced to denote the identical attribute of `messageevent` tag. The default value of this tag is `fipa` since this is the only type of message supported by Jadex.
- To denote `match` sub-element of `messageevent` element the `match` tagged value is introduced.
- The `posttoall` and `randomselection` boolean tagged values are introduced, to denote attributes of `messageevent` with identical name.
- The `abstract` and `exported` tagged values are introduced with identical usage as defined in extension section of DecidableGoal.

Constraints

There must be a Dependency with `usemessage` (defined in section 4.4.7) stereotype between AgentType and a CommunicationMessagePayload, or between EntityRoleType and a CommunicationMessagePayload, to denote that the agent or capability uses the specified Message Event for communication.

Rationale

Description of messages in Jadex has close semantics to CommunicationMessagePayload, where CommunicationMessagePayload are used to model objects transmitted by CommunicationMessages. Within CommunicationMessagePayload the Attributes specify the content of the messages, as defined in AML specification. The representation of this data is generated to a single Java class in form of fields, except for the attributes that have stereotype `parameter`, what represents parameters of a Message Event in Jadex.

The `content` parameter represents the data that is transferred by the Message Event, therefore when it is explicitly introduced, no data class is generated.

4.4 Additional Extensions

In this section we introduce additional extensions to AML used to model fundamental elements of Jadex, that have no semantically close representatives in AML.

We introduce Configure, Configuration, and Initialize stereotypes to model configurations of a Jadex agent. Parameter stereotype is used to generate parameters for various Jadex elements. The Beliefbase stereotype is used to extend Attribute to generate more specific Jadex beliefs. The Assign stereotype to model assignment of abstract elements. Additionally we define Use Message and Triggers connectors to model relations of Message Event.

We provide an informal definition of our extensions in additional Semantics section. Although the whole specification could be provided here, the Extensions section has remained for better readability.

4.4.1 Configure

Semantics

Stereotype: <<configure>>

Configure is a specialized Dependency (from UML) between a MentalSemiEntityType and a Configuration, used to model relationship between an AgentType and a Configuration or between an EntityRoleType and a Configuration.

Mapping

The Configure connector is used to map a Configuration to a `configuration` sub-element of `configurations` element in ADF.

Constraints

Source Code is generated only in case that Configure is between an AgentType and a Configuration, or between an EntityRoleType and a Configuration.

Rationale

Configure is used to model a relationship between an agent and its configuration, or between a capability and its configuration. In other words, it is used to specify the possible configurations of selected MentalSemitEntityTypes.

4.4.2 Configuration

Semantics

Stereotype: <<configuration>>

Configuration is a specialized Class (from UML) that represents a possible configuration of an AgentType or an EntityRoleType.

Mapping

Configuration is mapped to a `configuration` element to ADF, the name of the Configuration is mapped to the `name` attribute of `configuration` tag.

Extensions

One of the basic and important features that is provided by configurations is that the facts of beliefs and values of parameters can be redefined within a configuration. We model redefinition of these properties as follows:

- To model redefinition of a parameter of a goal, plan or a message event, a `parameter` (attribute with `parameter` stereotype) needs to be introduced within a configuration. The name of this parameter must have the form: name of the element that has the target parameter, and the name of the parameter separated by a dot. The new value of the result parameter is then generated from this parameter as defined in section 4.4.5.
- To model redefinition of a fact of a belief a `beliefbase` (attribute with `beliefbase` stereotype) needs to be introduced within the Configuration. The name of this `beliefbase` must have the form: name of the belief that has the target `beliefbase`, and the name of the `beliefbase` separated by a dot. The value of the result `initialbelief` or `initialbeliefset` is generated then from this `beliefbase` as defined in section 4.4.4.

The both cases are illustrated on an example in section 5.2.

Constraints

In both cases, for attributes with `parameter` stereotype and/or for attributes with `beliefbase` stereotype, the referenced elements have to be connected with Configuration using Initialize connector.

Rationale

An agent or capability in Jadex may have several configurations, that define the initial and end states of an agent instance; Configuration represent such a configuration, together with Initialize connector defines the set of initial and end elements of an agent or capability. Specifying parameter values or redefining facts of a belief are key aspects of Configuration, such mapping of attributes as defined above provides a simple way to capture this aspect.

4.4.3 Initialize

Semantics

Stereotype: <<initialize>>

Initialize is a specialized Dependency between a Configuration and one of MentalClass, EntityRoleType or CommunicationMessagePayload. It is used to model relation of specified elements with configurations, thus it specifies which element is initialized if an agent is instantiated or going to be terminated.

Mapping

The supplier of Initialize connector is mapped to:

- `initialgoal` or `endgoal` element of a configuration (depending on `inittype` tagged value, see extensions), in case that the supplier is a DecidableGoal. The name of the supplier is mapped to ref attribute of the generated element's tag. The initname tagged value is mapped to the name attribute of of generated element's tag.
- `initialplan` or `endplan` element of a configuration (depending on inittype tagged value), in case the supplier is a Plan. The name of the supplier is mapped to ref attribute of generated elements tag.
- `initialcapability` element, in case supplier is an EntityRoleType. The initialconfig tagged value is mapped to configuration attribute.The name of the EntityRoleType is mapped to `ref` attribute of initial capability.
- `initialmessageevent` or `endmessageevent` (depending on inittype tagged value), in case supplier is a DecoupledMessagePayload. The name of the supplier is mapped to ref attribute of generated elements tag.

In case supplier is a Belief `initialbelief` or `initialbeliefset` element of a configuration that is generated from attributes of the Belief. For how the facts of beliefs are redefined see mapping of element Configuration.

Extensions

- The `inittype` tagged value is introduced, with possible values of `init` and `end`, to denote when to initialize a component, thus when an agent is started or going to be terminated. The `init` value denotes that the supplier of Initialize dependency is instantiated when the agent is initialized.
- The `initname` tagged value is used to introduce a specific name for a goal instance that are specified in configurations.
- The `initialconfig` is used to specify the initial configuration of a capability that is initialized when an agent is started.

Constraints

Source code is generated only if the supplier of Initialization is one of the type: DecidableGoal, Belief, Plan, CommunicationMessagePayload, EntityRoleType.

Rationale

The Initialize connector is used to specify the initialized elements within a configuration.

4.4.4 Belief Base

Semantics:

Stereotype: <<beliefbase>>

Is a specialized Attribute (from UML) used to extend attributes, to generate `belief` and/or `beliefset` elements.

Mapping

Attribute with `beliefbase` stereotype is mapped to `belief` or `beliefset` element (in case the upper bound of attribute multiplicity not equals to 1), as following:

- The name of the attribute is mapped to `name` attribute of `belief` or `beliefset` tag.
- The type of the attribute is mapped to `class` attribute of the generated tag.
- If upper bound of the attribute multiplicity is 1, its initial value is mapped to `fact` sub-element of parameter.

Extension

- To model facts that are introduced as expression we introduce `expressionfact` tagged value that is mapped to `fact` or `facts` element depending on Attributes multiplicity.
- In case the upper bound of attribute not equals to 1, values can be specified by `initialfactlist` tagged value separated by "|" character.
- The `evaluationmode` tagged value is introduced to denote wheter the value of the fact is static or needs to be evaluated (it is dynamic). Its possible values are `static` and `dynamic`. It is mapped to identical attribute of `fact` or `facts` sub-element of `belief` or `beliefset elements`.
- To denote how often a fact needs to be evaluated the `updaterate` tagged value is introduced. It is mapped to `updaterate` attribute of `belief` or `beliefset` tag.
- The abstract and exported tagged values are used as defined first in section 4.1.1

Constraints

This stereotype can be applied only on attributes of a Belief.

Rationale

The `beliefbase` stereotype extends Attribute of UML, to specify more detailed beliefs within an ADF.

4.4.5 Parameter

Semantics

Stereotype: <<parameter>>

Is specialized Attribute (from UML) used to extend attributes, to generate Jadex specific parameters.

Mapping

Attribute with `parameter` stereotype is mapped to Jadex `parameter` or `parameterset` element (in case the upper bound of attribute multiplicity not equals to 1), as following:

- The name of the attribute is mapped to `name` attribute of `parameter` or `parameterset` tag.
- The type of the attribute is mapped to `class` attribute of the generated tag.
- If upper bound of the attribute multiplicity is 1, its initial value is mapped to `value` sub-element of parameter.

Extensions

- The `direction` tagged value is introduced to denote `direction` attribute of `parameter` or `pamameterset` tag.
- In case the upper bound of attribute not equals to 1, values can be specified by `initialfactlist` tagged value separated by "|" character.
- To model values that are introduced as expression we introduce `expressionfact` tagged value, that is mapped to `value` or `values` element depending on Attributes multiplicity.
- The `evaluationmode` tagged value is introduced to denote wheter the value of a parameter is static or needs to be evaluated (it is dynamic). Its possible values are `static` and `dynamic`. It is mapped to identical attribute of `value` or `values` sub-element of `parameter` or `parameters` element.
- To denote how often a value needs to be evaluated the `updaterate` tagged value is introduced. It is mapped to `updaterate` attribute of `parameter` or `parameterset` tag.

Rationale

The `parameter` stereotype extends Attribute of UML, to specify Jadex like parameters, that are commonly used by elements like Goals, Plans and/or Message Events.

4.4.6 Assign

Semantics

Stereotype: <<assign>

Assign is a specialized dependency between two DecidableGoals, between two Beliefs, or between two CommunicationMessagePayloads. It is used to model that an abstract element defined in a capability is assigned from an element defined in an agent, or in an other capability.

Mapping

This connector is mapped to an `assign` sub-element to the element that is generated from the dependent DecidebaleGoal, Belief, or CommunicationMessagePayload.

Constraints

An element can be assigned to only one abstract element, but more elements can be assigned to an abstract element (see [1]) .

In case of beliefs, all attributes (with beliefbase stereotype) of the dependent Belief, must have corresponding abstract attributes in the supplier Belief. (In this case the `belief` and `beliefset` elements are generated from the beliefbases of the Belief.) The concrete and abstract beliefs should not be mixed, thus one Belief should have only abstract or only not abstract (concrete) beliefbases.

Rationale

When an abstract element is defined within a capability, its body is assigned from a concrete element of an agent (or a capability). However an abstract element could be assigned, or specified, by more agents. Thus to model a concrete element we use a separate element that is assigned to an abstract element, using this Assign dependency. To clarify this situation see example in section 5.3.

4.4.7 Use Message

Semantics

Stereotype: <<usemessage>>

Uses is specialized Dependency between a MentalSemiEntityType and a CommunicationMessagePayload, used to model that the CommunicationMessagePayload is used by the MentalSemiEntityType, for various (not further specified) purposes.

Mapping

The default usage of this connector is defined in constraints section of CommunicationMessagePayload. Here we define only additional extensions for certain situations.

Extensions

The following extensions used to capture the case when Message Event is exported from a capability and included to an agent. The MentalAssociation between AgentType and CommunicationMessagePayload may have the following tagged values:

- `fromcapability` boolean tagged value (in case its value is true) denotes that the corresponding message event is included from a capability, or it is assigned to an abstract message event of a capability.
- `capabilitylocalname` tagged value is introduced to denote the name of the

capability from which the specified element is included.

Constraints

Source code is generated only in cases when the MentalSemiEntityType is an AgentType or an EntityRoleType.

Extensions are defined in order to facilitate code generation in situation:

- Where a concrete message event (the imported element) is imported from a capability that exports it. In this case the concrete element is connected using Use Message dependency both with the EntityRoleType (the capability where the element is defined), and to the AgentType (or EntityRoleType; where it is imported).
- Where a message event is assigned to a corresponding abstract element of a capability. For further description see section 4.4.6.

The `capabilitylocalname` tagged value is introduced since there could be a situation when an imported element is associated with more capabilities that are included to the agent (or capability). By defining this tagged value we avoid the possible ambiguities.

To clarify these extension see example in section 5.3.

Rationale

This connector plays a similar role as MentalAssociation, thus describes relations with a reusable element and an Agent or Capability. Events or message events are key features in BDI model, also in Jadex. However in ADF message events are specified in a general way, thus the concrete usage or behavior facilitated by a message events is not described (the only exception is when they serves as plan triggers, see next section).

4.4.8 Triggers

Semantics

Stereotype: <<triggers>>

Trigger is specialized Dependency (from UML) between a Plan and a CommunicationMessagePayload, used to model a relation that describes that the client, the CommunicationMessagePayload, serves as a trigger for the supplier, a Plan.

Mapping

Trigger is mapped to a `messageevent` sub-element of `trigger` element of a plan.

The name of the CommunicationMessagePayload is mapped to `ref` attribute of `messageevent` tag.

Rationale

Connector trigger is used to model that a Plan is reacting on a specified Message Event, this is a common and natural behavior of Jadex, since the message events are handled or processed inside a plan.

4.5 Summary

In this section we introduced a mapping of AML elements, mostly from Mental package to Jadex source code fragments, also extension to AML elements, in form of tagged values, which allows a generation of high detailed ADF. In addition other artifacts, skeletons of Plan implementations, and Java classes that represents data types that are transmitted in form of messages events, can be generated. With Mental package of AML, and few elements from other packages, using our simple extensions the major portion of BDI concept implementation in Jadex can be modeled, therefore can be transformed to corresponding source code representation in Jadex, thus ADF.

We introduced additional stereotypes as extensions that serve to model features that are more Jadex specific, like agents configurations, which can be generated with using Configure, Configuration and Initialize stereotypes, Jadex like parameters with Parameter stereotype, and some Jadex specific connectors related to Message Events, etc.

Other elements like internal events, imports, properties that are used in special cases (GUI update, etc) is out of scope of our discussion. Additionally we do not introduce extension for every type of plan trigger, and plan parameter mapping, that are used in some special cases, and for some more elements. However our goal with extension was not to generate a complete code, or Agent Definition Files, but to allow capture the major aspects of the system, thus allow generation of high detailed code. However with more additional extensions a complete generation of ADF could be reached, but a definition of such extensions and their mapping is out of limits of this thesis.

5 Examples

In this section we provide examples of code generation based on mapping, defined in previous section. First we introduce a model and describe how it is translated to target source code. Because of the simplicity of generated Java classes, we introduce only generated Agent Definition Files. Every example captures an aspect of agents definition. The not related elements are excluded from the diagram, as not relevant connectors, or tagged values that are not specified.

5.1 Elements of an Agent and Mental Relations

In our first example we describe an agent that is partially inspired by the "Cleanerworld" example (see [10]). The first diagram (Figure 12) describes an agent that has two goals, first is to maintain its battery loaded and the second is to clear a room. It has a Belief that tells the agent if there is a waste left in the room, and a plan for cleaning a room. The mapping of the elements on the diagram is defined in sections 4.1.1, 4.1.2, 4.1.3.

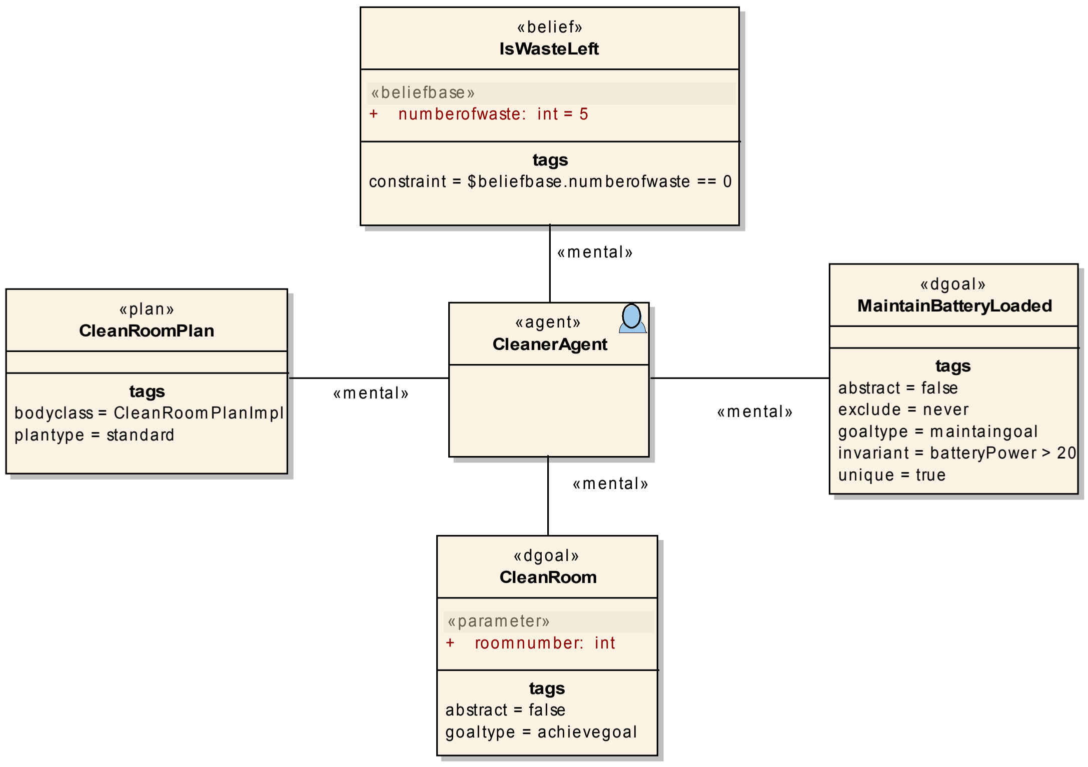

Figure 12: Defining elements of an agent

On the second diagram (Figure 13) the mental relations are modeled. The mapping of these

Contributions is defined in section 4.1.5 (cases 1,3,4). The model contains tree types of contribution. The Contribution between two goal denotes an inhibition, thus the `MaintainBatteryLoaded` goal inhibits the `ClearRoom` goal if its maintain condition (invariant tagged value on diagram, see section 4.1.6) not holds.

The Contribution between `CleanRoom` goal and the `CleanRoomPlan`, denotes that the goal serves as a trigger for the specified plan.

The third contribution between `IsWasteLeftBelief` and `CleanRoom` decidable goal, where the value beneficiaryConstrainKind (not shown on diagram) of the Contribution is post that is resulted in target condition of the CleanRoom goal is ADF.

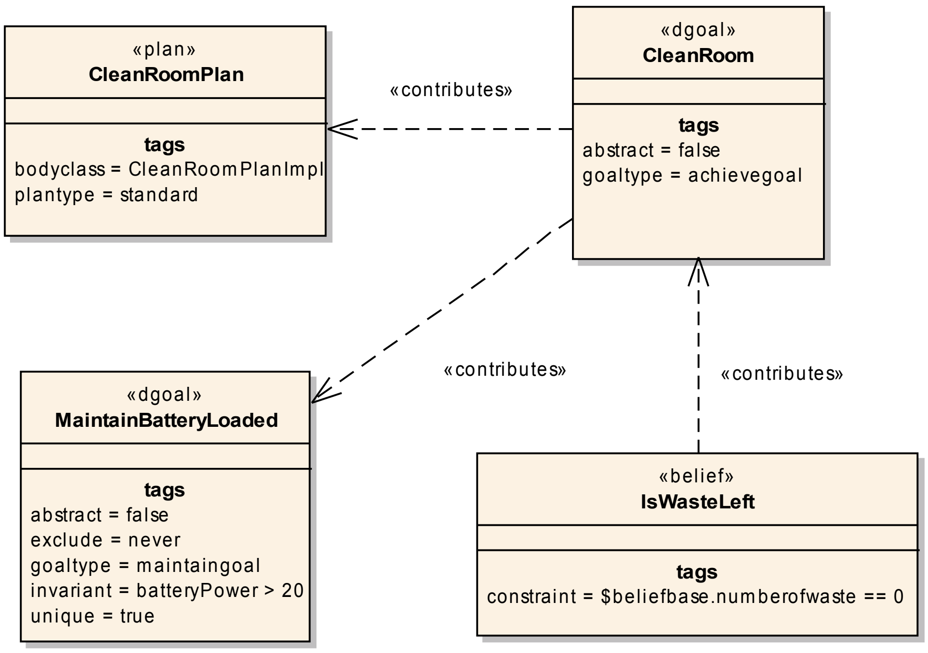

Figure 13: Mental relations

The generated ADF can be seen on Figure 14.

```
<agent ... name="CleanerAgent">
    <beliefs>
        <belief name="numberofwaste" class="int">
            <fact>5</fact>
        </belief>
    </beliefs>
    <goals>
        <achievegoal name="CleanRoom">
            <parameter name="roomnumber" class="int"  >
                <value></value>
            </parameter>
            <targetcondition>
                $beliefbase.numberofwaste == 0
            </targetcondition>
        </achievegoal>
        <maintaingoal name="MaintainBatteryLoaded" exclude="never">
            <unique/>
            <deliberation>
                <inhibits ref="CleanRoom"></inhibits>
            </deliberation>
            <maintaincondition>batteryPower>20</maintaincondition>
        </maintaingoal>
    </goals>
    <plans>
        <plan name="CleanRoomPlan" >
            <body type="standard" class="CleanRoomPlanImpl"/>
            <trigger>
                <goal ref="CleanRoom"/>
            </trigger>
        </plan>
    </plans>
. . .
</agent>
```

Figure 14: The generated Agent Description File

5.2 Configurations

The diagram in our second example describes a configuration of the agent introduced in the first example. The `DefaultConfiguration` of the agent initializes the `MaintainBatteryLoaded` and `CleanRoom` goals when an agent is created (tagged values of Initialize dependency are not shown on diagram). The configuration specifies a value for parameter `roomnumber` of `CleanRoom` goal, and a specifies an initial fact for `numberofwaste` belief. The mapping of the elements is defined is sections 4.4.1, 4.4.2, 4.4.3. The generated source code of ADF can be seen on Figure 13.

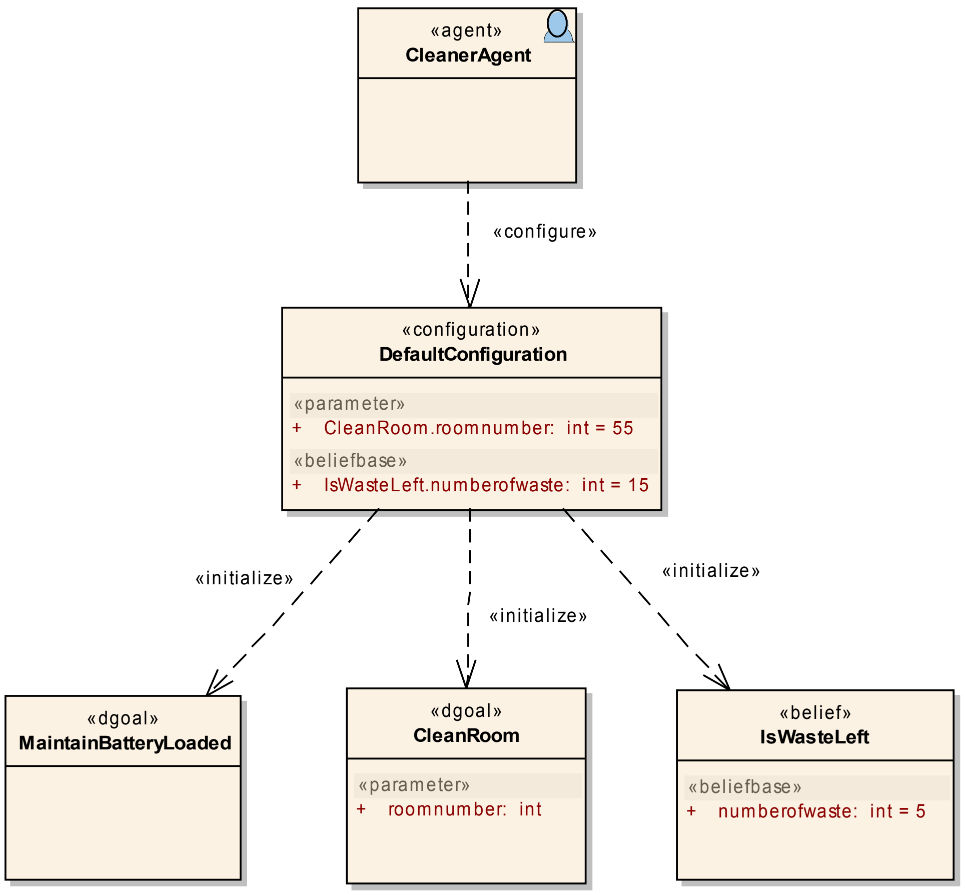

Figure 15: Agents default configuration

```
<agent ... >
...
   <configurations>
      <configuration name="DefaultConfiguration">
         <beliefs>
            <initialbelief ref="numberofwaste">
               <fact>15</fact>
            </initialbelief>
         </beliefs>
         <goals>
            <initialgoal name="InitialMaintainBatteryLoaded"
               ref="MaintainBatteryLoaded">
            </initialgoal>
            <initialgoal name="InitClearRoom" ref="CleanRoom">
               <parameter ref = "roomnumber">
                  <value>55</value>
               </parameter>
            </initialgoal>
      </configuration>
   </configurations>
</agent>
```

Figure 16: The generated configurations element in ADF

5.3 Include from a Capability

Our last example (Figure 17) illustrates the situation when some elements of an agent are included from a capability. There are two possible forms of include, their description and mapping are defined in sections 4.1.4, 4.4.6 and 4.4.7.

In the first case, the `AbstractGoal` goal is abstract within the capability. When an element is abstract within a capability, its definition or body is assigned from a concrete element, in our example from the `GoalToAssign` element, using Assign dependency. The tagged value `fromcapability` of mental association between `Agent1` and `GoalToAssign` denotes that the goal is assigned to an element in this case to `AbstractGoal` is included from a capability.

In the second case, when the specified element, on our example the `ConcreteMessage`, is defined within a capability and included as a concrete element to an agent. In this situation the value of `fromcapability` tagged value of `usemessage` dependency between `Agent1` and `ConcreteMessage` also needs to be true, and the `fromcapability` tagged value needs to be specified. In this situation the `ConcreteMessage` element is defined within the capability (as denoted using MentalAssocioation), and it is only referenced from Agent1 using `concrete` element.

In both cases there must be a PlayAssociation between the AgentType and the EntityRoleType. The result source code of the generated agent and capability can be seen on Figure 18 and Figure 19.

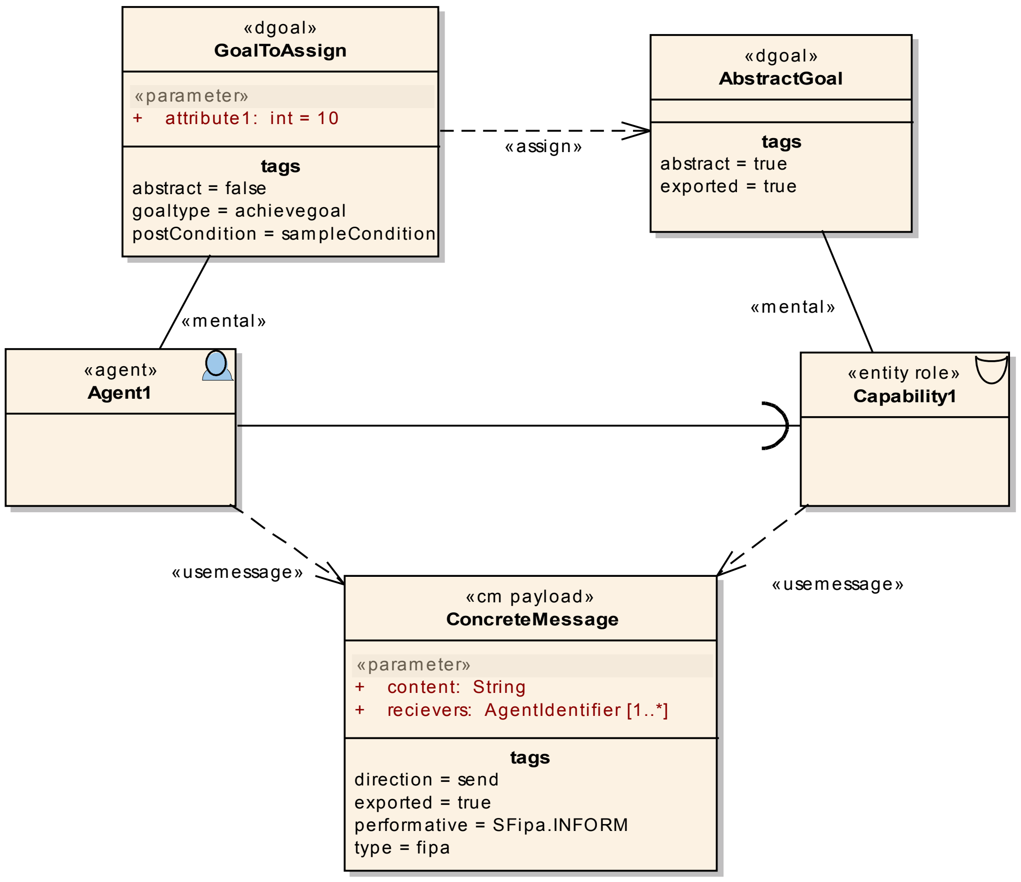

Figure 17: Example of includes from capability

```
<agent ... name="Agent1">
    ...
    <capabilities>
        <capability name="Capability1" file="packagename.Capability1"/>
    </capabilities>
    ...
    <goals>
        <achievegoal name="GoalToAssign" >
            <assignto ref="Capability1.AbstractGoal"/>
            <parameter name="attribute1" class="int"  >
                <value>10</value>
            </parameter>
            <targetcondition>sampleCondition</targetcondition>
        </achievegoal>
    </goals>
    ...
    <events>
        <messageeventref name="ConcreteMessage" exported="true" >
            <concrete ref="Capability1.ConcreteMessage"/>
        </messageeventref>
    </events>
    ...
</agent>
```

Figure 18: The generated agent

```
<capability ... name="Capability1">
    ...
    <goals>
        <achievegoalref name="AbstractGoal"  exported="true" >
            <abstract/>
        </achievegoalref>
    </goals>
    ...
    <events>
        <messageevent type="fipa" direction="send" name="ConcreteMessage"
                exported="true" >
            <parameter name="performative" class="String"
                    direction="fixed">
                <value>SFipa.INFORM</value>
            </parameter>
            <parameter name="content" class="String" direction="fixed" >
                <value></value>
            </parameter>
            <parameterset name="recievers" class="AgentIdentifier"
                    direction="fixed">
                <value>FirsdAID</value>
                <value>SecondAID</value>
            </parameterset>
        </messageevent>
    </events>
    ...
</capability>
```

Figure 19: The generated capability

6 Description of the Implementation

Code generations from models is one of the key features of Model Driven Engineering. Implementation in this case denotes a realization of code generation from models. In this section we provide an overview of approaches which can be followed by implementation, and introduce our implementation that generates code from AML to Jadex, based on our theoretical explorations introduced in section 4. Our solution follows an approach that generates source code from XMI files, therefore we also present a more detailed analyzes of this approach.

6.1 Overview of Approaches

Code generation from models can be implemented by following one of the architectural approaches presented in this section. Each of them has their advantages and disadvantages, we introduce only a general overview, a detailed analyzes can be found in [15].

Using CASE Internal Tools

This approach uses tools that are built in a CASE tool. A common idea behind CASE tools is that the functionality of the tool should cover the projects life-cycle, thus the code generation, also reverse engineering etc. One of the disadvantages of this approach is the tight integration with the CASE tools, also that these generators are designed to generate code for general cases, and simple patterns.

Add-in Producing Intermediate Language

The common scenario for this approach is that we implement code generation using Add-in for a CASE tool that produces intermediate language. The Add-in uses the tool's API that provides access to a model. The intermediate language is then transformed to target platforms code. Both code generation steps should be simpler than a direct translation. This approach is much more flexible than a direct translation, however a significant disadvantage is that an intermediate language has to be designed which should separate the frontend and backend generators in a well-balanced manner.

Add-in Producing Target Language

This approach is very similar to the previous one only the intermediate language generation step is excluded, thus no intermediate language has to be defined, the code is generated directly to target system. However this gives up also the flexibility which separated a part of the implementation from CASE tool. The implementation of this approach usually demands a huge amount of work.

Exported XMI

Code generation from XMI documents is the last approach that we mention here. It is very flexible way that allows us to separate completely implementation from CASE tools. This approach will be described in detail in the following sections.

6.2 Code Generation From XMI

In this section first we give a short introduction to XMI document format, then examine its usability for code generation.

6.2.1 The XMI Document format

The XML Metadata Interchange (XMI) is an Extensible Markup Language based standard for exchanging meta-data information. It is an international standard created by OMG, it can be used to express any meta-data whose meta-model can be expressed in MOF [43]. The design of XMI follows the vision of OMG, thus data can be separated to abstract models and concrete models [42]. Theoretically the most common use of this format would serve as an interchange format for UML models, or in some cases also for diagrams by Diagram Interchange (DI, XMI[DI]) language. However the current situation is that there are several incompatibilities between tools supporting XMI, thus implementation of serialization in these tools rarely follows the standard strictly. The format of the produced document in most cases it adjusted to tools needs. Additionally the usage of Diagram Interchange is almost zero. Thus the usage of XMI for exchanging models and/or diagrams between modeling tools is rarely possible [40].

Another issue with XMI is that there has been several versions created: 1.0, 1.1, 1.2, 2.0, 2.1. Additionally versions 1.* and 2.* are widely different. The most recent version is 2.1.1 released in 2007, for additional information see XMI specification [42].

Although there are lots of problems with usage of XMI in practice, due to the flexibility and separateness, that is allowed by this format, there is a strong effort for its wider utilization. As wee will see although in majority of cases it fails as an interchange format between CASE tools, but it is usable for other purposes, like code generation.

6.2.2 Realization of code generation from XMI

Code generation from XMI is a flexible approach of code generation that uses serialized UML models. A theoretical or ideal scenario of this approach is that the model is created with a CASE tool, which is saved to a XMI file, then the source code and other artifacts are generated from XMI with a code generator that handles it, as depicted on Figure 20.

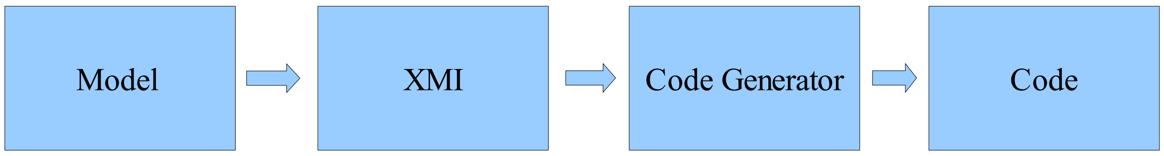

Figure 20: Ideal scenario of code generation from XMI

However in practice this scenario cannot be accomplished due to the problems which occur with the XMI standard. As described above CASE tools has questionable support for XMI in sense that the serialization not follows precisely the standard. In fact one of the biggest challenges that needs to be solved in this approach is handling dialects of XMI documents. In practice there must be a transformation from the XMI created by a CASE tool, to the XMI format which is supported by code generator. This requirement is mostly fulfilled by the code generator (see [39], [38]), or can be solved by providing a proper

XSL transformation, but this could mean a lot of work. The scenario according to these changes is illustrated on Figure 21.

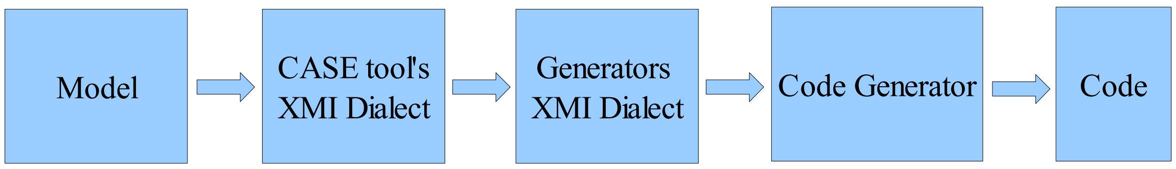

Figure 21: XMI code generation

We can now summarize the advantages and disadvantages of this concept.

Advantages:

- Complete independence from CASE tools.
- Code generators have easy to use and general API, in sense that arbitrary target language can be easily generated.

Disadvantages:

- Some CASE tools don't event support serialization models to XMI.
- Transformation of XMI dialects needs to be solved.
- However there are lots of opensource generators that support this approach, but lots of them is abandoned project or poorly documented, see section 6.3.
- Within a new version of a CASE tool the format of the produced XMI can be changed, therefore transformation that solves the dialect problems needs to be adjusted.

6.3 Code Generation Supporting Tool Overview

There is a huge amount of open source projects that supports code generation from XMI. In this section we give an overview about the most prominent and promising ones. In addition we introduce some supplementary solutions that might facilitate the generation process.

6.3.1 Code Generation Frameworks

Hereinafter we describe a few prominent representatives of code generation frameworks. Code generation portals lists a relatively large set of systems that supports the MDA paradigm using XMI files. Additional tools can be found for example at [37] or other code generation portals, however lots of them like - AXgen [36] or Butterfly [35] - are abandoned projects, or do not fulfill our requirements, namely that a code generator should enable us to generate code to an arbitrary language (in our case XML and Java), and it should support XMI conversion from various dialects, too.

- Eclipse Modeling Framework (EMF) [34] – is a very prominent modeling framework and a code generation facility that uses model specification in XMI format (to be more precise a format that follows XMI 2.1), and provides its general purpose object model that serves as a base for a variety of modeling tools. EMF has its own implementation of MOF [43] that follows the OMG standard architecture, includes lots of components that provide various modeling services such as validation framework, model queries, model comparison and transformation. EMF itself contains more projects that implement code generation like Xpand or JET.
- Java Emitter Templates (JET) [33] – is a code generator, part of the EMF, more precisely, part of the Model To Text project [32]. It provides template based code generation engine. The templates use a JSP like syntax, which provides a generic approach; it thus can be used to generate code to an arbitrary language. However JET has a questionable documentation.
- Acceleo [39] – is an easy to use code generation framework based on EMF. It uses templates and an own template language that allows us to generate code to an arbitrary target language. Acceleo is well documented, and also solves conversion of XMI dialects to the requested format. More information about supported CASE tools can be found in [9]. It provides an own object model for XMI 1.1 and 1.2, as complement to EMF object model that supports only XMI 2.x like formats.
- AndroMDA [38] – is a code generation framework that adheres to MDA paradigm. It is based on Netbeans MDR but also supports EMF repository implementation. Uses Apache Velocity as a template language for code generation, supports XMI conversion, but lacks good documentation.

We give our reference implementation (section 6.5) using Acceleo, since it is well documented, supports XMI conversion. It is easy to use, and it is based on EMF, thus after XMI file transformed to EMF format it becomes compatible with all EMF features. Acceleo has good integration with Eclipse IDE, provides an intuitive and easy-to-use perspective. AndroMDA is a good competitor of Acceleo, however it is more robust and as we mentioned relatively poorly documented, what also affected our decision.

6.3.2 Supplementary technologies

In this section we present some technologies related to our approach of code generation:

- XMI2 [31]– is a conversion tool implemented a service. It is designed to solve interoperability problems between CASE tools caused by XMI dialects, thus transforms XMI from one CASE tool specific format to an other. The actual implementation suffers from limitations, it doesn't support complex data types or UML profiles, also only a small amount of CASE tools is supported.
- MOF Model To Text Transformation Language [30] – is an OMG specification introduced in 2008, that defines a template based approach or a template language that translates a model to various text artifacts. This specification tries to avoid the appearance of new template languages with same purpose, and create a standard for template based code generator tools. The support of this language was announced also in Acceleo for the next major release [39].

6.4 Code Generation Using Acceleo

Acceleo [39] is a code generation frameworks that generates code from UML models serialized to XMI documents. It is designed to support XMI 1.* and 2.* formats, and to ensure compatibility with main UML modelers, therefore implements conversion of XMI dialects to appropriate format. Works with any meta-model, implementing MOF and QVT recommendations as specified by OMG [58]. This allows instant use of the new UML versions or any other meta-model. Acceleo supports template based code generation, it is independent from the target technology, thus is able to generate source code to any textual format, like Java, XML, C or C#. Supports integration with Eclipse IDE, that provides features that facilitates the development process, like meta-model and script based completion or real time error detection.

In this section we give an introduction of code generator development process with Acceleo, thus we examine its components and features that we used in our implementation.

6.4.1 XMI Compatibility

Acceleo is based on Eclipse Modeling Framework, therefore the XMI 2.* formats are directly supported. However EMF cannot work with XMI 1.* so it provides a bridge to overcome this problem. Currently the following combination of standards is supported, as defined in Acceleo User Guide [9].

XMI\UML	1.3	1.4	2.0
1.1	yes	yes	
1.2	yes	yes	
2.0			yes

However, this table is more like an orientation, since XMI 2.1 is also supported with UML 2.* meta-models, also it depends on the conversion that is made by Acceleo on XMI dialect. A general rule is that UML 2.* meta-models do not work with XMI 1.* and UML 1.* meta-models do not work with XMI 2.*.

The list of tools that produce XMI which can be used, thus converted by a bridge, can be found in Acceleo User Guide.

6.4.2 Templates

Templates are text files that are used to examine the model and to extract required information from it; they are used to produce result source code. In template we define a metamodel that specifies the type of the model, we introduce scripts that are applied on an arbitrary meta-class, on a UML Class, Attribute, Package or even the whole model. Scripts are responsible for the actual code generation. The last components of template files are services that are imported to solve complex operations, as described in next section.

```
<%
metamodel http://www.obeo.fr/acceleo/uml14
import myservices.MyServices
%>

<%script type="Attribute" name="ParameterGener"%>
<%if (multiplicity.range.upper != 1){%>
    <parameterset name="<%name%>">
    <%for (initialValue.body.split(";")){%>
        <value><%toString()%></value>
    <%}%>
    </parameterset>
<%}%>

<%script type="core.Class" name="sample" file="<%name%>.txt"%>
<plan name="<%name%>">
        <%for (feature.filter("Attribute")){%>
<%ParameterGener%>
        <%}%>
<trigger>
    <%for (supplierDependency.client[stereotype.name == "belief"]){%>
    <condition>
        <%taggedValue[type.name == "constraint"].dataValue%>
    </condition>
    <%}%>
</trigger>
</plan>
```

Figure 22: Sample template file

A sample template can be seen on Figure 22, when we use the UML 1.4 metamodel, as defined by `metamodel` keyword, to generate a simple XML files from Classes. In fact we generate Jadex like plans as root elements, using the second script. As can be seen on this example, functions that are used to access model elements, all keywords and functions are encapsulated within "<%","%>" special brackets, other text is emitted to result source code without any change. Every script has its name, and can be referenced from an other, in our example the first script with `ParameterGen` name is used to generate `parameterset` elements from Attributes of the class. As defined by `type` keyword it is applicable only on

UML Attributes. We check if the Attributes upper bound don't equals to 1, and if not assume that initial values of attributes contains more values, composited into one string and separated by “;” character, therefore we use a service that splits the initial values into parts, and we generate `value` elements for each of these parts, using for loop.

The `file` attribute of the script specifies the name of the result file that is created with the script. In our case the name of the plan will be equal to the name of the Class, and the result will be a “txt” file.

Additionally we generate condition triggers from Beliefs (element that has stereotype “belief”) that are connected with this class using dependency connector, thus our class is the supplier of the dependency, and the Belief is the client. Belief's constraint is a tagged value that filtered from other tagged values with expression introduced within “[ ]” brackets.

This short example gives an introduction to the structure and basic components of the templates. For more detailed description please refer to the Acceleo User Manual.

6.4.3 Services

Services provide complex operations that would be complicated to realize within a template file. Services are written in Java programming languages and are implemented as operations of a Class file, thus we are able to handle complex problems easily. This approach provides an access to low level EMF classes that represents the model. Services or operations can be called from template files, the first parameter of the operation is always the current generation node. Node can be an arbitrary object type including primitive Java types; `EObject` - which is an EMF equivalent of `java.lang.Object`, or `ENode` which is an abstract data type defined by Acceleo - , used to encapsulate a value.

We distinguish between two types of services:

- Services that are automatically integrated to template files. A complete description of these can be found in Acceleo Reference [8].
- Utility services that are integrated to generation modules and are imported by templates.

An example usage of a string service can be seen on Figure 8, which is applied on initial values of attributes, that splits the initial string values to parts and returns them as members of a list that are handled then by a for-cycle.

6.4.4 Execution Chains

An execution chain serves to group several operations on models, it is like a “Makefile” or “Ant” script for traditional development. An example use case for execution chains is to define application of more templates for a single model, therefore to it simplifies the process of code generation execution, or launching bulk operations. The Eclipse integration for Acceleo provides a GUI that enables an easy-to-use editing and launching of these chains (see [9]).

6.5 Implementation Details

Our implementation consists templates that generates source code from AML using extensions that are defined in section AML to Jadex Mapping. This is a reference implementation that also serves as complementary information for mappings.

We provide our templates for UML 1.4 metamodel. Because of conventions of Accele introduced in section 6.4.1, our implementation is suitable for XMI 1.* files.

As it is described in section 6.5.2, due to the problems with Acceleo bridge that handles XMI transformations, the realization of generation using UML 2.* metamodel and XMI 2.* files wouldn't be even possible. However XMI 1.* format is supported by most of the CASE tools that implement serialization of models to XMI. Therefore from practical viewpoint this not means strong restriction. On the other hand an implementation for UML 2.* metamodels based on our implementation could be realized easily, possibly by a simple transcription, since the difference between templates would be only in manner that the model elements are accessed within a template.

The three templates we provide are:

- Template that generates Agent Description Files, thus both for agents and capabilities.
- Template that generates skeletons of Java classes implementing Jadex plans.
- Template that generates implementation of a Java classes that represents data specified by CommunicationMessagePayloads.

In addition we provide implementation of an XSL transformation that solves bug that rises with the transformation of XMI files when the multiplicity of an attribute is unlimited, see section 6.5.2.

As supplementary work, we provide implementation of AML profile including our extension for Sparx Systems Enterprise Architect [29]. We used this CASE tool for creating our models and testing the templates, since it is well known, widely used tool and has a very good support for UML profile implementations. It supports serialization of models for both XMI 1.x and 2.x formats. Enterprise Architect is supported by XMI code generation frameworks like Acceleo and AndroMDA. Although it is not free, has a relatively low price. We tested our generator with models exported in version 7.1.

6.5.1 Description of Templates

The two templates that generates Java classes are very simple. These files are generated from one class following a simple mapping. Templates are applied on UML Classes, and the target Plan and/or CommunicationMessagePayload is filtered depending on stereotypes, additionally in case of CommunicationMessagePayloads also filtered depending on its attributes.

On the other hand the generation of Agent Definition Files, that are the central targets of our code generation, is more complex. We use same the template for generating the ADF for both agents and capabilities, since these files has identical structure and identical elements. Depending on stereotypes (`<<agent>>` or `<<entity role>>`) only the name root tag generation is different. However if the constraints defined in section 4 are followed in the model, the generated code will be a valid ADF.

The detailed description of the template file is out of the scope of this document. We will describe here our policy how the template file is structured or what guidelines were followed during its development. Additionally we describe a set of problematic situations that are not trivial to handle. Since Acceleo templates are easily understandable in most cases, and are also self explaining, therefore detailed description is not even necessary.

The general structure of ADF is described in section 3 (see Figure 6). Our effort was to design templates to be easily understandable and readable, however due to the characteristics of template language in some situations is hard to achieve this goal. We divide our template to more scripts. Scripts are useful both from point of view of structuring a template, and they creates a reusable part of a template. Thus a script can be called from other scripts. In our case complex elements like definitions of goals, plans, message events are generated using a separate script. Another good example where the re-usability of scripts can be illustrated are scripts that generates `parameter` elements, which are reusable for generation of parameters for goal, message event, or plans. The implementation of mapping is in most cases straightforward, the only difficulties we experienced were caused by properties of Acceleo template language.

On Figure 23 is depicted a small fragment of the template that calls the `ExtendedGoalGener` script which implements generation of goals from a Class. We check if the class has `dgoal` stereotype and whether it is associated with the agent with a `mental` association. The script is called with two arguments, which are tagged values of the association. These values can be then accessed from the script.

This example illustrates a problem that arises with the Acceleo template language when some complex situations need to be handled or more detailed constraints need to be checked. The source of the problem is that a `for` loop, or elements inside of a `for` loop have only local scope. Thus no external elements can be accessed inside a loop, just elements of the list on which the loop is applied. Therefore in our example without passing parameters to script, or without using a script, the tagged values `fromcapability` and `capabilitylocalname` of the mental associations could not be accessed. The solution for such situations is to call for loop on association and pass the tagged values as parameters for the script. However as can be seen, this property makes the language less usable in complex situations.

```
<goals>
    <%for (association[association.stereotype.name=="mental"]){%>

    <%inverseAssociationEnd().participant[stereotype.name =="dgoal"].
        filter("Class").ExtendedGoalGener(
    association.taggedValue[type.name=="fromcapability"].dataValue,
    association.taggedValue[type.name=="capabilitylocalname"].dataValue

    )%>
    <%}%>
</goals>
```

Figure 23: Call of the script that generates various goal elements

As can be seen from this example the properties of the template language implied that a complex pattern matching is realizable only using services.

6.5.2 Problems with XMI Transformations

In Acceleo documentation [9] is introduced a list of CASE tools that are supported, or XMI files produced by these tools can be imported and used for code generation. Although Enterprise Architect is supported by framework we experienced problems with the transformations of XMI files.

The biggest restriction that is identified also in Acceleo User Guide is that actually the framework can handle only Class and Deployment diagrams. Elements from Activity diagrams cannot be transformed, therefore cannot be used for code generation. For this reason in profile implementation we define AML Plan as a Class not as an Activity.

During the development and testing we identified that the models serialized to XMI 1.2 are transformed properly to requested XMI format. The only bug we identified with this conversion is that the bridge cannot handle unlimited upper multiplicity of Attributes. To handle this problem we introduced an XSL Transformation that converts the standard unlimited multiplicity notation "*" to "-1", what is the corresponding representation in Acceleo metamodel implementation.

With transformation of XMI 2.1 (only XMI 2.1 is supported) files suffers from following problems:

- Associations are not transformed, thus during the XMI transformation the associations are excluded. However a solution of this bug is provided by the community. (see community pages at [39])
- Dependencies are not transformed properly.
- If there are stereotyped attributes the transformation cannot be accomplished.

These restrictions makes the XMI 2.1 created by Enterprise Architect unusable for us.

7 Possible Extensions

In this section we introduce complementary solutions that are close to the subject-matter introduced in this thesis.

7.1 Reverse Engineering

Reverse engineering can be defined in very general sense; in our context it covers the process of transforming source code artifacts to a higher level abstraction, to UML models.

Beside to provide code generation, CASE tools are usually designed also to handle reverse engineering from selected programming languages. The fact that the support of CASE tools in great majority is constrained for only a number of programming languages, thus no general reverse engineering mechanism is provided, makes these tool from our aspect less comfortable. A list or comparison of CASE tools that examines also this functionality is provided by the community at [28].

It is good to realize that in our case, to accomplish reverse engineering based on the mappings defined in section 4 in major case the Agent Definition Files – which are XML documents – needs to be taken into account (The only one exception is when a data class is generated from CommunicationMessagePayload, the fields of this file are not represented in ADF).

Some CASE tools like Visual Paradigm for UML [27] provide functionality of reverse engineering of XML documents, however this functionality is not common for most of CASE tools, also the class diagram we got after reverse engineering excludes duplicated elements, for example multiple beliefs. Therefore the proper result for us would be an object diagram, which could be transformed to a requested AML model by a “model to model” transformation language like ATL [26].

Another possible approach which is closer to our implementation is to use XMI documents also for reverse engineering. A possible scenario is depicted on Figure 24, that converts source code to an XMI document, which is then translated to an another XMI dialect that can be imported by the target tool. One crucial step that is mandatory in most cases is the translation of XMI files, what is the same problem we experienced at code generation (section 6.2) except that code generators doesn't support reverse transformation.

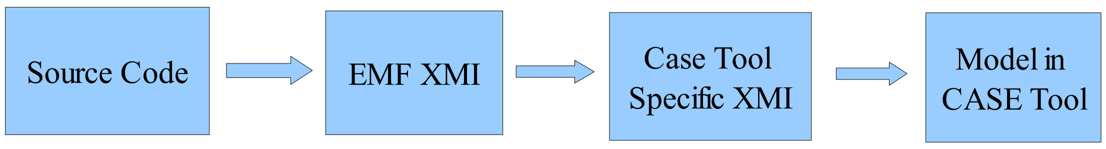

Figure 24: Reverse engineering scenario using XMI

The other crucial step is to get the XMI file from source code. A good candidate that could solve this problem is the project called MoDisco [25], which is part of EMF, and provides capability of reverse engineering from XML files, that could be saved after transformations to XMI files, what is a basic functionality of EMF. However the project has poor documentation and it is in early development state, some of its components are just in state of incubation.

At present EMF lacks support for reverse engineering, however it supports a simple but complete API that can be used for creation and manipulation of EMF models from Java code. Additionally a DOM parser can be used to transform XML documents to object trees. Traversing trees representing ADF, a model can be produce using EMF API that could be also saved to XMI by EMF. However a detailed analyzes and design of such application or this approach is out of the scope of this thesis.

7.2 Source Code Generation from Standard UML Models

Source code generation from UML stance for translation of a higher more abstract representation of a system to a lover level abstraction or language. Jadex uses Agent Definition Files that provides relatively high level of abstraction of logic of an agent in the system. The mental package of AML provides language constructs that are able to represent these files or this description of logic, therefore it is very suitable for generation of these files, as described in section 4. The other central components of Jadex are plans, which are implemented in Java language. As described in section 3 plan is an arbitrary Java class that extends either `jadex.runtime.Plan` or `jadex.runtime.MobilePlan` classes. In this section we present a short introduction to the possibilities and limits of source code generation from UML diagrams to lover level languages like Java, therefore for plan implementations.

7.2.1 Class Diagrams

Code generation from UML class diagrams is a common practice. The majority of UML tools provides this functionality, since the purpose of these diagrams is to describe the structure of the classes within the system. An introduction how a class diagram is translated to the target language can be found in [22]. However by using such a translation only the skeleton of the classes is generated, i.e. class definition, its fields and methods. But not implementation of methods or behavior of the system.

7.2.2 State Machine Diagrams

The UML state machines package defines [56] a set of concepts that can be used to modeling a behavior using a finite state transition system. In most cases state machines are drawn for a single class to show its behavior during its lifetime [12].

A default approach that is followed by the code generators is to represent the state machine with executable code in the target language, i.e. classes, attributes, operations that implements its functionality with easily extensible architecture.

A realization of this concept is provided in Sparx System's Enterprise Architect [29] that implements the state machine within one class, it generates enumerations, attributes and operations, that enables effective execution of state behaviors for simple state machines. It generates executable code in such a manner, that only operations that represents activities in states needs to be completed (for details see [3]).

Another, a more complete implementation of this concept is provided in IBM Rational Rhapsody [24]. Or in addition described by Iftikhar Azim Niaz in his paper [5] and in his dissertation [4]. These solutions generate executable source code representation of more complete state machines, including constructs like composite states and fork/join, however the result code becomes much more complex.

Although it is possible to generate a code that captures the general semantics of state machines, its usage in practice is questionable. The more precise semantics implied by the context or by the characteristics of system within it is applied, therefore the required corresponding implementation could be different.

In association with Jadex plans, from semantic view, state machines could be more suitable for description of plans, but not for mobile plans, since as specified in [1] mobile plans are stateless and its behavior is determined by actual parameters, therefore their characteristics contradicts with fundamental characteristics of state machines.

7.2.3 Activity Diagrams

Activity diagrams are used for description of lover level behaviors, like algorithmic behavior or control/object flow models. In practice, from the viewpoint of code generation, activity diagrams have strong limitations. To describe a detailed source code, e.g. a complex algorithm, Activity diagrams requires as large amount of nodes that simply becomes unpractical as expressed by Chaves in [19].

Although it is used to generate code fragments or skeleton of an algorithm. A typical implementation is provided in Enterprise Architect, that generates (possible recursive) if-then-else statements from decision nodes, or while loops in case of cyclic graphs [3].

Jadex mobile plans as denoted in [1] in most cases typically have if-then-else block structure, therefore code generation from activity diagram could be more suitable for them.

7.2.4 Evaluation

In previous section we described the diagram types that theoretically may serve as source for code generation. Activity and State Machine diagrams are behavioral diagrams that could be used, in theory, to generate corresponding complete implementation. However as we've seen this goal is not fulfilled, in fact to create these two diagram instances only with purpose of code generation, could lead to unwanted or unnecessary amount of work.

Other type of diagrams like Sequence diagrams are identified by some software engineers useless for this purpose [19], although some tools like Enterprise Architect implements code generation from Sequence diagrams, and there are a few cases when it is applicable and produces negligible amount of code snippets, see [3].

8 Conclusion and Future Work

During the work we fulfilled our objectives to define a mapping between AML and Jadex, what is the theoretical background for code generation. We also specified an extension to AML, which allows us to denote more Jadex specific features and model most of the BDI aspects of Jadex. In addition we created an extension for capturing elements of Jadex, namely configurations, that is out of the scope of BDI model, but plays an important role in specification of Agent's mental aspects. The main targets of our discussion were the relation between Mental package of AML and the Agent Definition File of Jadex. Agent Definition Files describe the mental aspects of agents following the DBI paradigm. We showed that a highly detailed ADF file can be generated from models, using Mental package of AML (and a few more elements from other packages) and some simple extensions. In addition we defined mapping also on low level artefacts of Jadex, like implementations of plans. Our intent was not to generate a complete code, or Agent Definition Files, but to allow to capture the major aspects of the system, that supports a generation of high detailed code. However, with more additional extensions a complete generation of ADF could be reached, although a definition of such extensions and their mapping is out of limits of this thesis, but could be a topic of some possible future work.

In the second part of the thesis we described our practical work, the implementation of code generation, that generates source code from XMI files. Although this CASE tool independent approach theoretically has big advantages, but its implementation and the quality of the tools that support it is currently questionable - both from the viewpoint of CASE tools and the code generation frameworks. In other words, the biggest insufficiency of this approach is caused by the fact, that the XMI documents produced by the tools rarely follow strictly the standard. On the other hand these frameworks, in our case Acceleo, provide a simple template language. This language allow a high-level access to models, and generates arbitrary target source code. This makes the implementation phase relatively simple. This means that implementation of a good defined mapping is relatively straightforward. A possible future practical work, based on the theoretical part of our work, would be an implementation of reverse engineering. Thus creation of models from Agent Definition Files, what we mentioned also in section 7.1.

Generally, the theoretical part we see as the main contribution of this thesis. We provided theoretical foundations for code generation from AML to Jadex; or in a more abstract view, how a code generation can be realized from high-level model elements like elements of AML Mental package.

9 Appendix

List of Figures

Reference List

[1] Alexander Pokahr, Lars Braubach. Jadex User Guide. 2007
http://ignum.dl.sourceforge.net/project/jadex/jadex/0.96/userguide-0.96.pdf

[2] Jadex homepage
http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

[3] Geoffrey Sparks. Enterprise Architect User Guide. 2009
http://www.sparxsystems.com/bin/EAUserGuide.pdf

[4] Iftikhar Azim Niaz. Automatic Code Generation FromUML Class and Statechart Diagrams. 2005

[5] Iftikhar Azim Niaz, Jiro Tanaka. Code Generation From UML Statecharts.

[6] Ján Danč. Formal Specification of AML. 2008

[7] Jiří Bělohlávek, Petr Knoth. Multi-agent programming. 2007
http://www.stud.fit.vutbr.cz/~xknoth00/resources/jadex.pdf

[8] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe. Acceleo Reference. 2008
http://www.acceleo.org/doc/obeo/en/acceleo-2.6-reference.pdf

[9] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe. Acceleo User Guide. 2008
http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf

[10] Lars BraubachAlexander Pokahr. JadeX Tutorial. 2007
http://ovh.dl.sourceforge.net/project/jadex/jadex/0.96/tutorial-0.96.pdf

[11] Lin Padgham, Michael Winikoff. Prometheus: A Methodology for Developing Intelligent Agents.
http://www.cs.rmit.edu.au/agents/Papers/aose02.pdf

[12] Martin Fowler, Kendall Scott. UML Distilled Second Edition A Brief Guide to the Standard ObjectModeling Language. 1999

[13] Michael Bratman. Intention, Plans, and Practical Reason. 1987

[14] Michael Wooldridge, Nicholas R. Jennings, David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. 2000
http://www.csc.liv.ac.uk/~mjw/pubs/jaamas2000b.pdf

[15] Michal Kostic. AML Code Generation. 2006

[16] Radovan Červenka, Ivan Trenčiansky. Agent Modeling Language - Language Specification. 2004

[17] Radovan Červenka, Ivan Trenčiansky. The Agent Modeling Language. 2007

[18] Radovan Červenka, Ivan Trenčiansky. The Agent Modeling Language - A Comprehensive Approach to Modeling Multi-Agent Systems. 2007

[19] Full code generation from UML class, state and activity diagrams
http://abstratt.com/blog/2007/06/01/full-code-generation-in-uml-from-the-class-state-and-activity-diagrams/

[20] Raquel Trillo, Sergio Ilarri and Eduardo Mena. Comparison and Performance Evaluation of MobileAgent Platforms.
http://sid.cps.unizar.es/PUBLICATIONS/POSTSCRIPTS/ICAS07.pdf

[21] Richard Evans, Paul Kearney, Giovanni Caire. Message: Methodology for Engineering Systems of Software Agents. 2001.

[22] Robert C. Martin. UML Class Diagrams Part 1 - Class Diagrams. 1997
http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf

[23] Viviane Torres da Silva, Carlos J. P. de Lucena. MAS-ML: a multi-agent system modeling language. 2003

[24] Rhapsody homepage
http://www-01.ibm.com/software/awdtools/rhapsody/

[25] MoDisco homepage
http://www.eclipse.org/gmt/modisco/

[26] ATLAS Transformation Language at Eclipse.org
http://www.eclipse.org/m2m/atl/

[27] Visual Paradigm for UML homepage
http://www.visual-paradigm.com/product/vpuml/

[28] List of Unified Modeling Tools at Wikipedia.
http://en.wikipedia.org/wiki/List_of_UML_tools

[29] Sparx System - Enterprise Architect homepage
http://www.sparxsystems.com/

[30] MOF Model to Text Transformation Language at OMG
http://www.omg.org/spec/MOFM2T/1.0/

[31] XMI2 Homepage
http://modeling-languages.com/content/xmi2-tool-exchanging-uml-models-among-case-tools

[32] EMF Model to Text homepage
http://www.eclipse.org/modeling/m2t/

[33] Java Emitter Templates homepage
http://www.eclipse.org/modeling/m2t/?project=jet#jet

[34] Eclipse Modeling Framework
http://www.eclipse.org/modeling/emf/

[35] Butterfly Code Generator homepage
http://butterflycode.sourceforge.net/

[36] AXgen Hompage
http://axgen.sourceforge.net/

[37] Collection of MDA tools at Modelbased.net
http://www.modelbased.net/mda_tools.html

[38] AndroMDA homepage
http://www.andromda.org/

[39] Acceleo homepage
http://www.acceleo.org

[40] XML Metadata Interchange

http://en.wikipedia.org/wiki/XML_Metadata_Interchange

[41] Catalog of OMG Modeling and Metadata Specifications
http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[42] MOF 2.0/XMI Mapping, Version 2.1.1. 2007
http://www.omg.org/technology/documents/formal/xmi.htm

[43] MetaObject Facility at OMG
http://www.omg.org/mof/

[44] Model Driven Architectures homepage
http://www.omg.org/mda/

[45] Publicly Available Agent Platform Implementations at FIPA homepage
http://www.fipa.org/resources/livesystems.html

[46] Cougaar homepage
http://www.cougaar.org/

[47] Grasshopper - A Universal Agent Platform Based on OMG MASIF and FIPA Standards
http://cordis.europa.eu/infowin/acts/analysys/products/thematic/agents/ch4/ch4.htm#5

[48] JADE homepage
http://jade.tilab.com/

[49] Agent Commnunication Language Specification homepage
http://www.fipa.org/repository/aclspecs.html

[50] OMG's Mobile Agent Facility homepage
http://www.omg.org/technology/documents/formal/mobile_agent_facility.htm

[51] OMG MASIF homepage
http://www.omg.org/cgi-bin/doc?orbos/97-10-05

[52] Foundation for Intelligent Physical Agents homepage
http://www.fipa.org/

[53] Agent UML homepage
http://www.auml.org/

[54] Agent-Object-Relationship (AOR) Modeling and Simulation homepage
http://oxygen.informatik.tu-cottbus.de/aor/?q=node/1

[55] Tropos homepage
http://www.troposproject.org/

[56] OMG Unified Modeling Language (OMG UML),Superstructure. 2009
http://www.omg.org/cgi-bin/doc?formal/09-02-02.pdf

[57] Unified Modeling Language at OMG
http://www.uml.org/

[58] OMG Homepage
http://www.omg.org/

Abstract

Vývoj a údržba komplexného multi-agentového systému je veľmi zložitý problém, a je výzvou pre softvérové inžinierstvo. Analýza, dizajn, implementácia, testovanie a prevádzka takýchto systémov môžu byť veľmi ťažko realizovateľné. Agent Modeling Language (AML) je komplexný agentovo-orientovaný modelovací jazyk, ktorý slúži na zachytenie rôznych aspektov multi-agentových systémov, a tým uľahčuje ich vývoj. V tejto práci implementujeme generátor kódu, ktorý produkuje zdrojový kód z modelov AML do agentového systému Jadex. Definujeme mapovanie z AML do Jadex. Zavedieme jednoduché rozšírenie pre jazyk AML, ktoré umožňuje generovanie vysoko detailného zdrojového kódu. Implementujeme generovanie kódu pomocou frameworku Acceleo, ktorý umožňuje generovanie kódu nezávisle na CASE nástrojov, prístup ktorý na tento účel používa XMI súborov.