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

Code generation from AML
Implementation into CASE tools and support for existing agent platforms

Master's Thesis

Michal Kostic

Informatics

Thesis advisor:
Mgr. Radovan Cervenka

Bratislava 2006

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

Bratislava, May 2006

.....
Michal Kostic

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

Contents

1 Introduction 6
1.1 Abstract 6
1.2 Introduction to the Agent Development 6
1.3 Objectives and Tasks 7
1.4 Work Structure 7
2 Sources of Inspiration 8
2.1 Code Generator as a Compiler 8
2.2 Model Driven Architecture 9
3 Technological Backgrounds 11
3.1 UML Metamodel Implementations 11
3.2 Agent Platforms 12
3.2.1 Platform for Distributed Communication 12
3.2.2 Behavior and Message Based Platform 12
3.2.3 Message Handler Based Platform 12
4 Architectural Approaches 13
4.1 Scenario 1: Add-in Produces Intermediate Language 13
4.2 Scenario 2: Add-in Produces Java Code 14
4.3 Scenario 3: Using CASE Internal Tools to Generate AP Code 14
4.4 Scenario 4: Export to XMI 14
4.5 Conclusion 16
5 CASE Tool Evaluation 17
6 Agent Platform Evaluation 21
7 Overview of AML 25
7.1 Introduction to AML 25
7.2 AML and UML 25
7.3 Description of AML Elements 26
7.3.1 Architecture Package 26
7.3.2 Behaviors Package 28
7.3.3 Examples 29
8 Overview of JADE Platform 32
8.1 Introduction 32
8.2 Runtime Environment 32
8.3 Programming Model 33
8.3.1 Agent 33
8.3.2 Behavior 35
8.3.3 ACLMessage/MessageTemplate 36
9 AML to JADE Mapping 37
9.1 Mapping Definitions 37
9.1.1 Agent 37
9.1.2 RoleType 37
9.1.3 PlayAssociation 38
9.1.4 BehaviorFragment 39
9.1.5 AgentType Communication 39

9.1.6 Communicative Interaction 40
9.1.7 InteractionProtocol 40
9.1.8 AgentExecutionEnvironment 40
9.1.9 HostingProperty 41
9.1.10 HostingAssociation 41
9.1.11 ServiceSpecification 41
9.1.12 ServicedElement 42
9.1.13 ServicedProperty 42
9.1.14 ServicedPort 42
9.1.15 ServiceProvision 42
9.1.16 ServiceUsage 43
10 Implementation Description 44
10.1 Architecture 44
10.2 Design and Implementation 44
10.3 Algorithms and Complexity 46
11 Conclusion and Future Work 48
12 Appendix 1 49
12.1 List of Figures 49
12.2 List of Tables 49
12.3 Reference list 49

1 Introduction

1.1 Abstract

The code generation is a technique for transforming analytical models into the engineering artefacts. In this thesis I will present the implementation of the code generator that translates AML models to the code executable on JADE agent platform. The code generator architecture is based on the pipe model of the conventional compiler structure. The execution process and control flow of code generation is influence by Model Driven Architecture. It uses two phases in code generation: a generation of Platform Specific Model from Platform Independent Model in first phase and generation of target code from the Platform Independent Model in the second phase.
In order to enable the implementation I also show mapping between AML and JADE, evaluate available CASE tools and agent platforms.

1.2 Introduction to the Agent Development

Multi-agent systems have developed from the theoretical and experimental stage to the business-ready technology. This is also putting a pressure on standardizing development of multi-agent systems in the same way the conventional software development processes are standardized. As the response to this need many agent development methodologies have been defined. For example TROPOS [Bresciani 2002], MaSE [DeLoach 1999], MESSAGE [Evans 2001] and others.
Modeling on different levels of abstraction plays important part in all of the main methodologies. This task is covered by multiple modeling languages that are often defined as part of the methodology. One of the latest additions to agent modeling languages is AML that I will be using throughout this work. It is UML based agent modeling language trying to unify the best practices from other modeling languages and methodologies.
The next step from modeling is implementation of agents. It is usually realized in some of the agent platforms. Current agent platforms (like JADE [TILab 2006], Grasshopper [IKT++ 2006], JACK [JACK 2006]) are mostly implemented as Java frameworks that provide classes and services used to implement agents. Implementing agent then means implementing a Java class that inherits from some of the platform base classes.
Despite the agent platforms provide framework that makes agent development easier and more straightforward, there can be certain amount of code overhead needed to perform basic tasks like sending of messages. This extra work can be overcome by using the tool that transforms a model to the target code – a code generator.
Code generation is well known technique used to transform models (e.g. UML) to the executable artifacts (source code). Also in the multi-agent systems area some code generators were implemented for various agent modeling languages and target platforms.
Specifically for AML there exists a commercial implementation that transforms AML to the code of proprietary agent platform. Despite this fact we think that it is important for making AML more widely accepted and support is to provide code generation tool that would be based on free or open source technologies.

This work introduces an implementation of new code generator that transforms AML model in StarUML to the JADE code. Both of the platforms (StarUML and JADE) are open source projects.
The core objective while implementing the code generator was exploring and defining the mapping between AML and JADE concepts. The complexity of this a task is implied by the gap between the modeling concepts and implementation classes.
As supportive work I will provide a brief evaluation of CASE tools and agent platforms where I will justify the selection of StarUML as source CASE tool and JADE as target agent platform. The work also includes a discussion on the code generator structure in relation to compiler and model driven architecture, introduction to AML and JADE platform in scope necessary for understanding the code generator.
If you are looking for a definition and explanation what agent is you should consider consulting some specialized literature on Multi-Agent Systems or artificial intelligence. In this text I will consider as an agent any class that is marked by proper stereotype in UML or instance of AgentType metaclass in AML. Any other properties of agent I leave as intuitive. However it is worth noting that some of the typical (but not required or limited to) features of multi agent systems are proactivity, concurrency, complex interactions, asynchronous messaging, distributed architecture etc.
Same, this work is not intended as advocacy of Multi-Agent Systems but should provide a view on how Agent Modeling Language may be supported by CASE tools.

1.3 Objectives and Tasks

The main goal of this work is implement code generation tool for the selected platform. It should support the widest range of AML possible.
The task necessary for achieving this target is defining mapping between AML and JADE. It has to be explored what mappings are possible and what are the limitations of both AML and agent platform.
Before actual implementation work the target platforms will have to be chosen. There is need to evaluate both the source CASE tool possibilities and target agent platform because currently there are many implementations that vary in quality, support and availability.

1.4 Work Structure

This work is structured into 10 main chapters each dealing with a specific part of the problem. The chapters 2 and 3 introduce methodical and technological grounds that will be used in the rest of the work. In chapters 5 and 6 I will evaluate available CASE tools and agent platforms and choose candidates that will serve as platform for the code generator. Chapters 7 and 8 introduce actual technologies that are used for code generation – namely JADE and AML. The core chapters are 4, 9, 10. In chapter 4 I discuss few architectural options for implementing code generator and reason about choosing one approach. Chapter 9 is crucial for code generator implementation. It defines the mapping between AML and JADE concepts. Chapter 10 explains the design and implementation issues of code generator.
Chapter 11 concludes the work and chapter 12 provides lists of references.

2 Sources of Inspiration

In the following section I will look at the code generation from the two perspectives. One is older approach of generating code from high level programming languages to the machine code. The other is modern Model Driven Architecture (MDA) approach that translates UML models to high level programming languages. Both of these approaches will serve as inspiration for the architecture and implementation of the AML code generator.

2.1 Code Generator as a Compiler

The task that complier usually handles is translating language on the higher level of abstraction (e.g. a programming language) to the language on the lower level of abstraction (e.g. a machine code). This task is similar to code generation from AML model (high level of abstraction) to agent platform code (lower level of abstraction). Since compilers are well explored and developed technology mapping of a code generator to compiler will help in effective structuring the code generator.

Note that in this chapter I will use term "code generation" as substitute for "code generation from AML/UML to target agent platform code". By compiler I will mean the compiler that translates conventional programming language (like C++ or Pascal) to machine code.

I will now show how to map general compiler structure to code generator structure.

General compiler structure is as follows [Aho 85]:

- Lexical analyzer
- Syntax analyzer
- Semantic analyzer
- Intermediate code generator
- Code optimizer
- Code generator

I expect reader to be familiar with these concepts. Should you have any questions, consider consulting mentioned literature.

Each of the concepts can be mapped to the code generator component. Following section will show how to perform the mapping.

- Input language – in code generator input language is some kind of graphical language as opposed to programming language in the compiler. The main difference is that while programming language can be described by a context-free grammar, modeling languages are usually described only semi-formal description. This results into structure being too complex for description by context-free grammar. I will use UML/AML as the foundation of our code generator.
- Lexical analyzer – lexical analyzer recognizes tokens as the smallest parts of programming language. In code generation tokens are single model elements. The elements are atomic so there is no need for lexical analyzer but application that provides access to the model elements can be viewed as simple lexical analyzer. This application can be for example extensibility API in CASE tools or UML metamodel implementation.

- Syntax analyzer – In traditional compiler implementation the syntax is defined by formal grammar. Language described by the grammar is recognized by parsing algorithm like CYK, LALR parsers etc. But UML/AML structure is too complex for description by context free grammar. Thus I will give up the standard parsing algorithm and use algorithm that traverses graph consisting of elements and relations between them and recognizes patterns along the way.
- Semantic analyzer – consistency checks should be performed on the graph of recognized patterns. They can be implemented in this phase
- Intermediate code generator –using the compiler idea of platform independent intermediate language can be used in code generation as well. The intermediate language can take various forms. One candidate for intermediate language is UML model that uses special profile that represents concepts in lower level of abstraction close to the target agent platform. Another option is set of objects that represent these concepts. Generally intermediate language can be seen as platform specific language from the MDA (see next section).
- Code optimizer – In compiler structure the code optimizer tries to perform transformations which will result in the better performing target code. In code generator it could perform optimizations on the intermediate language. These optimizations could be enhancing structure of the model by applying design patterns or optimizing message number in interactions.
- Code generator – the resulting artifact of compiler is machine code. In the case of code generator the role of lower level language takes target agent platform code. It is usually in the form of plain text files containing object oriented programming language or descriptor file (e.g. in XML). The source for the generation of code is same as in compiler an intermediate language.

The mapping between compiler and code generator shows that these concepts share very similar structure and mapping between them is straightforward. This allows me to use compiler structure as foundation for code generator architecture.

2.2 Model Driven Architecture

One of the currently most accepted approaches in code generation is initiative based on the OMG standards called Model Driven Architecture (MDA see [OMG 2006b] for homepage of MDA). I will introduce it here as the main source for architectural consideration and control flow of code generator.
The aim for defining MDA is to allow business allow businesses developing bespoke applications to concentrate on determining their business requirements for the application [Haywood 2004].
Technically MDA is a set of related standards specified by the Object Management Group (OMG). These standards UML, XMI, MOF, OCL, CWM (see [OMG 2006a] for details) are used to turn a model (usually in OMG's Unified Modeling Language [OMG 2005]) to engineering artifacts. Resulting artifacts can take form of either executables like source code or non-executables like documentation.

MDA does not only define technologies used for modeling and model transformations but also high level standard steps in code generation. They are defined in following order (see Figure 1):

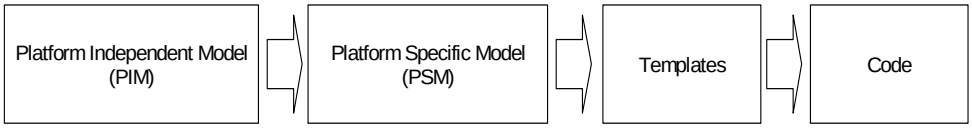

Figure 1: MDA Process from [Code Generation Network 2004]

First a source analytical model is created. It captures the business requirements in concepts close to the reality. In MDA the initial model is called Platform Independent Model (PIM). It specifies entities in the most abstract form possible while still capturing all of the requirements.
The next level down is the Platform Specific Model (PSM). This model is a transformed version of PIM, which adds all of the structures required to implement the PIM in the target platform. PSM is also modeled as a UML model. Note that different platforms require different PSM models.
The output code is generated from the PSM using predefined templates that transform PSM elements to the target code or other artifacts.
Main objective of PIM is to capture business requirements in the language using concepts close to reality. This makes it close to the standard understanding of analytical model.
PSM is more detailed in terms of technical implementation and its objective is to bring the business requirements closer to the target platform. With focus on the implementation issues PSM corresponds to the usual view of design model. Essentially PIM is analytical model of a domain; PSM is design model for specified platform.
Important issues in MDA modeling are transformations between PIM and PSM, and PSM and code. These translations are performed through transformation steps where patterns in one model are mapped to patterns in another model. Mappings may be done in a procedural way (i.e. by implementation of pattern recognition in conventional programming language – Java, C#…) or in a declarative way (i.e. by pattern transformation language).
OMG as founder of MDA is supporting Query-View-Transform (QVT) [OMG 2002] initiative. Its task is to describe transformations between two patterns described in MOF language. An interesting implementation is submission from QVT Partners [QVT 2003].
From the technical point of view MDA in its current state is a promising and intuitive idea that formalizes usual process of refining model expressed by analytical and design model. However tools for code generation are quite immature. Most of them allow only restricted model access, code generation from static structures, low level of customization, narrow concentration on one specific platform (EJB, database access …) etc.
I will not strictly stick to MDA but I will use it as a guideline for code generator architecture and recognition process.

3 Technological Backgrounds

After introducing two existing approaches that will be foundations for architectural considerations in code generator implementation I will discuss two technologies that are crucial for agent code generation.
First is UML metamodel implementation that defines how UML/AML model is programmatically accessed. Second technology briefly introduced here are agent platforms. They create environment for executing agents and will be a target platform for code generation.

3.1 UML Metamodel Implementations

UML metamodel implementation is a set of APIs used to access UML model from the programming language. It provides access to the model in terms of classes, associations, interactions and other UML elements. These elements are accessible as instances of metaclasses (e.g. Class from UML).
Metamodel implementations can be segmented to two groups according to their integration with CASE tool. One possibility is that metamodel implementation is included in CASE tool. Typically it is part of API which enables extensibility in the form of add-ins written in object oriented programming languages. This kind of metamodel implementation can be found in almost any CASE tool – either accessible via public API or as purely internal technology for handling UML model.
Other variation of metamodel implementation is standalone application. They can read UML model exported to the file (commonly XML/XMI) and provide access to it not in terms of XML nodes and elements but in terms of UML elements.
XMI (stands for XML Metadata Interchange) can be viewed as common interchange format for UML models. It is (as its name stands for) XML based language for interchange of metadata described by MOF (Metaobject Facility). MOF is OMG language for specification of languages like UML. XMI can be used to save any language defined by MOF but in its most prominent application it became industry almost-standard for exchange of UML model among different CASE tools.
Exchange works so that UML model is saved to the standardized format (XMI) which may be accessed by other CASE tools or UML metamodel implementations.

The quality of UML metamodel implementation is crucial for the successful and easy code generator implementation. It can be evaluated using following measures for the quality of UML model implementation:

- Implementation of all UML elements and their features
- easy navigability across UML connectors
- easy navigability across element references
- stable and well documented API
- strongly typed representation of UML model elements

Standalone UML metamodel implementation could be a good choice for implementing a code generator, but most of the available implementations are either internal projects of CASE tools development companies (and thus are accessible only as part of the CASE tool) or their documentation is at very low level (like NSUML [Novosoft 2002]).

This leads to the conclusion that code generator should use CASE tools API for accessing the UML model to avoid problems with unstable metamodel implementation.

3.2 Agent Platforms

The important MAS enabling technology is agent platform. It is application that simplifies the implementation of multi-agent systems through a middle-ware [TILab 2006]. Agent platforms usually provide a library of classes and a set of services that make development of agent applications easier. One of the agent platforms will have to be chosen as the target platform for code generation. Here will present three different approaches to agent platform and their representatives. The chapter Agent Platform Evaluation will discuss available implementation of each of the approaches.

3.2.1 Platform for Distributed Communication

First kind of agent platforms provides minimum services supporting agent development and uses coding techniques closest to traditional Java programming. Its programming patterns are very similar to traditional distributed communication technologies like RMI or IIOP.
For example agent can be implemented as a class running in its own thread that is able to communicate with remote classes via communication wrappers that make the distributed communication transparent to the client.
This kind of agent platform provides ease of development and well known environment but it does not bring any higher level of abstraction from the OO language to agent oriented language.
Example of this approach is Grasshopper [IKT++ 2006]

3.2.2 Behavior and Message Based Platform

To this category belong agent platforms that incorporate many advanced agent features including asynchronous messaging, Agent Communication Language message types, interaction protocols and ontology support. The trade-off for high level programming can be code overhead needed for performing actions like message sending.
Agent can be implemented as class with its own thread that schedules behavior classes for execution. Behaviors implement agent actions and message handling.
Programmers used for conventional object-oriented programming might not find this programming model completely convenient but this approach seems very promising for implementation of code generator because it works with terms that are close to the AML model.
Example of this approach is JADE.

3.2.3 Message Handler Based Platform

This concept represents theoretical model of agent platform where main agent logic is implemented as message handler of specified type of messages. It is easy to understand model because most functionality is grouped around message handler. At same time it might imply that process flow inside the agent can be unintuitive because it has to be translated to the terms of message handlers.
Example of message handler based approach is commercial product LS/TS by Whitestein Technologies.

4 Architectural Approaches

Code generator can be implemented using various architectural approaches. In this chapter I will summarize some them in the context of existing technologies and try to justify selection of one approach that I will use in implementation of code generator.

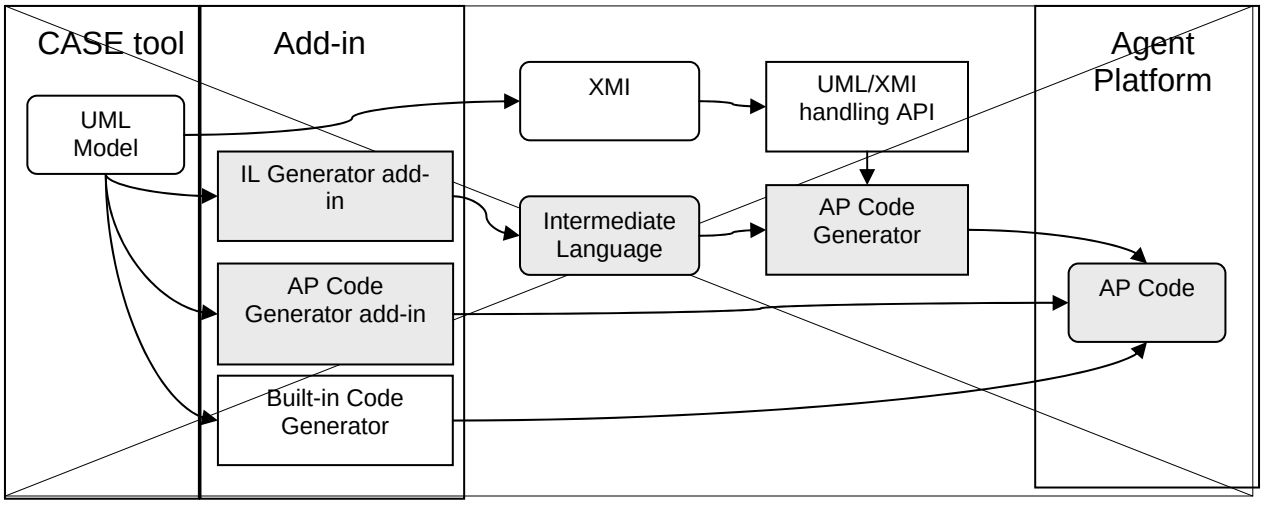

Figure 2: Possibiliteis of control flow in code generator

Figure 2 graphically describes possible approaches to code generation implementation. Rounded rectangles represent artifacts (like UML Model, XMI representation of model …) and square rectangles represent components (like intermediate language generator …). Gray rectangle means that this part of system has to be implemented. White rectangle, in contrary, means that there exists component implementing required functionality.
Generally each path from "UML Model" to "AP Code" represents one possible approach to code generation. I will describe them more closely.

4.1 Scenario 1: Add-in Produces Intermediate Language

Used path in Figure 2: UML Model → IL Generator Add-in → Intermediate Language → AP Code generator → Agent Platform Code

This scenario tries for flexibility in code generation yet utilizes existing components.
It uses CASE tool add-in to generate intermediate language (IL) from the AML model. Intermediate language is then converted to target agent platform language. It is not specified if model in intermediate language is explicitly expressed in form of text files or it is only a set of objects representing concepts of intermediate language.
Both transformations (AML → IL and IL → agent platform code) are supposed to be simpler and more straightforward then direct translation from AML to agent platform code.
IL generator and backend generator are kept separate so that this approach provides certain flexibility in creating variations for different CASE tools or target platforms and at same time helps isolating changes in both CASE tool and target platform APIs to small modules.

The drawback is that IL has to be carefully designed and developed to enable the mentioned separation of frontend and backend generators at reasonable level. This means creating language different from current standards. Implication is that there won't be any tools that would help handling IL.
Generally this scenario provides very good balance between amount of work needed for implementation, flexibility and reuse of current components.

4.2 Scenario 2: Add-in Produces Java Code

Used path in Figure 2: UML Model → AP Code generator add-in → Agent Platform Code

Scenario 2 for generation is very similar to the scenario 1. The difference is that it gives up the generation of intermediate language and directly translates AML to target agent platform language. This approach gets around the issue of designing IL with compromise on flexibility and separation from the existing applications as trade-off.
Also the transformations from AML to agent platform code will be less straightforward then in previous case.
It seems that these trade-offs are not worth the gain from giving up the definition of IL.

4.3 Scenario 3: Using CASE Internal Tools to Generate AP Code

Used path in Figure 2: UML Model → Built-in Code Generator → Agent Platform Code

Scenario 3 attempts to make the best use of existing code generation tools present in the CASE tool. It relies on the CASE tools original add-ins to generate code.
Despite contemporary CASE tools are equipped with code generation tools they are not expected to be used for agent generation. Their functionality is restricted to provide easy code generation based on predefined patterns. As result they don't provide any support for agent development.
In this case the direct translation from AML to agent platform code would be complex and hard to define using simple patterns that are available in the built in code generators.

4.4 Scenario 4: Export to XMI

Used path in Figure 2: UML Model → XMI → UML/XMI handling API → Agent Platform Code

Scenario 4 strives for maximum flexibility and independence from CASE tools which would allow using code generator for input from any CASE tool.
In this scenario the AML model exported to XMI file and then UML metamodel implementation is used to access this file and generate target agent platform code.
Despite this approach has theoretical advantages (flexibility, independence of CASE tools) it encounters problems with the current state of implementation of XMI handling APIs and adoption of XMI standard itself.
The problem of free standalone XMI handling applications is that they are immature and poorly documented.

XMI on the other hand is defined as standard but different CASE tools vary in quality of XMI export [Marchal 2004]. Each tool has specific variation points from the official standard when exporting UML model to XMI.
These technical issues make this scenario currently unsuitable but it might be interesting option in the future when XMI handling frameworks get more mature.

The Table 1 summarizes the advantages and disadvantages of the approaches.

Table 1: Scenario comparison

Scenario	Advantages	Disadvantages
Add-in produces intermediate language	• good coverage of AML/AP features • easier code generation from not-so-high-level language • back-end generator (intermediate language to AP) will not be affected by UML or CASE tool changes	• need to develop intermediate language (possibly language for AP design tools or Agent Common Layer) • need to generate intermediate language different from standards (maybe extension of XMI or a kind of an abstraction language which could be part of AP design tools) • need to translate AP intermediate language to Java by our own tools
Add-in produces Java code	• stable API for UML 2.0 available in modeling tools • good integration with modeling tool (EA)	• a lot of work has to be done, which will be tightly coupled with single tool • no back-end module makes generation for different platforms harder • hard to integrate with development environments like Eclipse in the future (hard reciprocal influence of code and model)
Using CASE internal tools to generate AP code	• reuse of existing tools (template framework) • tight integration with modeling environment (if designer changes model => it is easy for him or her to regenerate code	• tight integration with modeling environment (if CASE changes, code generator has to change too) • does not enable model/code cooperation on-the-fly – this approach can not ensure close interaction of code and models, • CASE tool code generators don't provide any support for generating agents
Export to XMI	• complete control over generated code • most flexible architecture	• almost complete code generation process coverage would have to be implemented which results into

		a lot of work to be done • dependent on XMI handling framework (currently are available open source frameworks but they are poorly documented) • questionable support for UML 2.0 • need to cope with non standard XMI • support to XMI standard is not fully adopted by CASE tool vendors

4.5 Conclusion

As stated, each the presented approaches to code generation has its advantages and disadvantages. There is no evident best approach that would exceed the others in all aspects. Considering advantages and disadvantages I consider approach of Scenario 1 using CASE tools add-in to generate an intermediate language as a good candidate for the architecture of code generator.

This approach, if carefully designed, provides a good balance between platform independence, reuse of existing tools and ease of programming. As mentioned, defining intermediate language can be non-trivial task. In order to avoid need of developing special IL and implementing tools handling it, IL does not need to be explicitly defined but can be represented by set of objects.

Platform independence of Scenario 1 approach is implied by the use of intermediate language. It allows easy retargeting of generated code by defining new PSM model and its transformation to code. However, some attention has to be exercised so that implementation of PIM is not too tightly coupled with CASE tool UML model implementation.

This approach makes use of existing and stable implementation of UML metamodel in CASE tools. UML handling API is used at traversing the model graph.

Note that ease of programming is dependent on the quality of UML handling interface which may differ significantly among different CASE tools.

5 CASE Tool Evaluation

As discussed in the chapter 4, the code generator won't use CASE tool only as a source of the AML model but CASE tool will be an environment for implementing code generator as add-in.
On the market there are many CASE tools of various qualities. There was need to choose one as a code generator platform.
First different CASE tools where long-listed based on the internet sources. Then five tools where short-listed based on the price, availability and external reviews like [Godfrey 2006] or [Eckel et al 2003].
Each of these tools was evaluated according to the following criteria important for code generator implementation.
Price – since project is not for profit oriented one of the main issues was price. CASE tool will be preferably freeware or open source software. If there was no suitable free tool available, relatively low price (<US$100) would be advantage.
UML Support –AML uses UML extensively and thus the proper support for UML 2.0 is vital. Otherwise many aspects of AML would be left unsupported or would have to be implemented using workarounds.
Extensibility – AML will be implemented using stereotypes of standard UML and code generation tool will be implemented as an add-in. As result the selected CASE tool has to support both of these features – add-in and stereotype customization. In order to facilitate code generation the CASE tool has to also provide API for accessing UML model that complies with quality requirements mentioned in chapter 3.1.
UML tools were evaluated from the personal experience and some information was added from the official documentation and other users experience expressed in the discussion forums.

- *Enterprise Architect* [SPARX 2006] - offers good features but it is not a free product. User reviews on enterprise architect are also favorable. The important positive feature for code generation is clear, easy to use and well documented UML model handling API. Despite Enterprise Architect contains many bugs it has good community support via discussion forums and is fixed builds are released quite often. The first version of code generator was implemented in this tool. Price: $95-$120
- *Rational Rose* [IBM 2006] – it is a famous CASE tool with powerful features and very good visual output quality. The drawbacks of this tool are non-standard behavior in certain cases, little user friendliness and very high price. Price: >$1000
- *ArgoUML* [Tigris 2005] – it is open source project. At time of writing UML support was limited to UML 1.4 only. The add-in interface and programming documentation are not clear. ArgoUML needs a lot of maintenance and improvement before it could be used as platform for code generation. Price: free
- *Poseidon* [Gentleware 2006] – provides good UML support features even in free version. However the free version is limited in number of diagrams and does not support add-in creation. Commercial version is slightly above Enterprise Architect but still reasonably priced. Price: $249
- *StarUML* [StarUML 2006] – is open source project which targets for full UML 2.0 support and good usability. It is based on the previously commercial tool and so despite it is in its first version it looks quite mature. The API is well documented and the UML profile

extensibility is good. There is also available community support via forums. The application is bound to Windows 2000/XP platform. Price: free

The concise list of features of CASE tools can be found in Table 2.

Table 2: CASE tool comparison

CASE Tool	*UML Support*	*Extensibility*	*Platform*	*Price*
Enterprise Architect	UML 2.0	Very good – profiles, COM add-in, user defined menus	Windows 98/2000/XP	$95-120
Rational Rose	UML 2.0 with specific deviations	Very good - profiles, COM add-in (Windows platform only), user defined menus	Linux/Solaris/ Windows 2000/XP	>$1000
Poseidon	UML 2.0	Very good - profiles, add-in, user defined menus	Java	$249
ArgoUML	UML 1.4 excluding sequence diagram (limited)	Poor – insufficient documentation to evaluate	Java	Free
StarUML	UML 2.0	Good – profiles, own icons for profiles, COM add-in, user defined menus, not user friendly API	Windows 2000/XP	Free

From the above results we can make following conclusions:
Among commercial tools Enterprise Architect provides very good value, performance and extensibility options. At same time it is the cheapest commercial CASE tool from the comparison.
StarUML is in par with Enterprise Architect in features even if its API is not that well structured as the one of Enterprise Architect. It is young open source project that is based on the originally commercial software. It has been undergoing heavy development that slowed down after publishing official first version. Despite risk of instability it has been chosen as the target platform.

6 Agent Platform Evaluation

Agent platform is essential part of multi-agent system which defines how the multi-agent application will be implemented.

There exists a multitude of agent platforms. I was focusing on the platforms that are free and have good community acceptance. This criterion is important for one of the goals of the thesis – to support usage of AML. Generator for quality and popular platform is expected to be better accepted and more widely used than generator for minor experimental platform.

With respect to this the main requirements for Agent Platform (AP) that would be chosen as a target platform for code generation add-in were set as follows:

Price – since project is not for profit oriented one of the main issues was price. Preferably freeware or open source software. In case there is no free tool available relatively small price (<US$100) would be advantage.

Support for agent development – agent development is a paradigm that extends usual object oriented programming and as such it requires different programming model. The closer the programming model of agent platform is to the AML model the easier will be the implementation of code generation. It will help also the mapping to be more straightforward and clear.

Some of the questions to evaluate this measure are: How close is the agent platform model to the agent/AML model? How much programming is necessary to implement agent features (message sending, mobility…)?

Ease of programming – the target platform for code generation should be easy to use and implement applications in.

Questions for evaluation are: How different from standard programming are the concepts used in the agent platform? How big is the code overhead (initializations, property settings…) involved in the programming of agent? Is there tool support?

Standards compliance – agent mobility and cooperation is important feature in agent development. It is expected that the agent platform will be interacting with other platforms and so interoperability is important feature. Standards compliance is directly supporting interoperability by enabling the different platforms to communicate on the common basis.

Some of the questions to evaluate this measure are: Is agent platform FIPA compliant? Does AP work on the open standards for message transmission? Are there other standards supported (XML, web services…)

Support for distributed computing – the support for distribution of agent platforms (or parts of agent platform) significantly eases development of distributed applications.

Questions for this measure are: Does AP support transparent distribution over multiple computers? Is there integrated support for agent mobility?

Practical usage – agent platform should be viable enough to be implemented in real-world projects. Otherwise it would be only experimental platform for which code generation might not be necessary.

Typical questions representing this measure are: What real-world projects have been implemented using this platform? Is there a community support?

Agent platforms were evaluated from the personal experience and some information was added from the official documentation and other users experience expressed in the discussion forums.

- *JADE* [TILab 2006] – is an open source platform that was developed by TILab. It is often evaluated as one of the best currently available open source agent platforms. The programming model is very close to the agent paradigm (using terms like Agent or Behavior). Downside is that it requires slightly higher programming overhead for operations with agents and communication. Despite this it has been used in many applications in various industries (telecommunications, healthcare, manufacturing…)
- *Grasshopper* [IKT++ 2006] – is also a free agent platform. It was developed by IKV++. Grasshopper is representative of agent platform where the programming concept is close to the conventional programming (does not use explicitly behaviours or messages). The programming concept of Grasshopper is built around distributed communication technologies like RMI or IIOP. This makes programming with Grasshopper easy and convenient when used as platform for distributed computing but it does not bring the improvement to the programming efficiency for truly agent oriented applications. Grasshopper was used in multiple applications mostly in telecommunications.
- *LS/TS* [Whitestein 2006] – is proprietary agent platform developed by Whitestein Technologies. Its programming is based on creating xml files for the agents that describe their structure and message handling capabilities. I have used it mostly for theoretical considerations. Real world applications were not published yet.

Concise list of agent platform features can be found in Table 3

Table 3: Agent Platform Comparison

Agent Platform	***Ease of programming***	***Support for agent development***	***Standards compliance***	***Distributed Programming***	***Platform***	***Price***
JADE	Retrieving agent references and sending/receiving messages includes certain amount of code.	Programming model very close to agent model. Works with terms Agent, Behaviour, Message.	FIPA compliant	If agent identifier is known there is no need to know exact location of agent. Migration of agent is explicit sending messages with specific mobility ontology to AMS or calling agent methods.	Java	Free
Grasshopper	Programming model close to Java with wrappers for distributed communication	Agents are basically Java classes that communicate via wrapper that provides uniform access to distributed communication technology (rmi, iiop, plain socket)	Requires add-in for FIPA compliant communication Implements OMG MASIF standard Supports range of distributed communication technologies (RMI, IIOP)	If agent identifier is known there is no need to know exact location of agent. Migration of agent is explicit using call to agent methods.	Java	Free
LST	Declarative way of programming – agent is defined by xml. Simple but not standard concept.	Programming model very close to agent model. Works with terms Agent, Behaviour, Message.	FIPA compliant		Java	N/A

From the above results we can make following conclusions:
JADE can be chosen as suitable platform. It is open source project that has been used in various practical applications. Its programming model is close to the agent oriented programming concepts. Despite it requires code overhead for performing basic tasks it is a flexible environment.

7 Overview of AML

In this chapter I will provide overview of Agent Modeling Language (AML) in the scope necessary for understanding and usage of the code generator. For detailed specifications please refer to the AML language specification [AML 04]
In the first part I will mention the definition of AML and motivation which lead to the definition of new language. Section AML and UML will put AML into the context of widely used modeling language UML. The last section – AML Elements description - will provide brief description of the elements that are used by the code generator.

7.1 Introduction to AML

AML Language specification [Cervenka et. al 2004] defines AML as follows:
"The Agent Modeling Language (AML) is a semi-formal visual modeling language for specifying, modeling and documenting systems that incorporate concepts drawn from Multi-Agent Systems (MAS) theory." Where semi-formal refers to "the language that offers the means to specify systems using a combination of natural language, graphical notation and formal language specification. It is not based on a strict formal (e.g. mathematical) theory."

Motivation for defining AML was to create a practically usable language that would be feasible for commercial software development. There exist multiple modeling languages focusing on modeling of agent oriented applications but the need for a new language was implied by the fact that current agent modeling languages suffer from various problems such as limited expressiveness, lack of documentation and questionable supportability by CASE tools. AML was intended to overcome these problems.

In order to make AML accepted and usable it was based on the various sources among which belong:

- UML 1.5 and UML 2.0
- OCL 2.0
- Various agent modeling languages and methodologies (e.g. MESSAGE, AUML…)
- FIPA standards
- Existing agent-oriented technologies
- Multi-agent system theories and abstract models

In this work I am using AML version 0.9.

7.2 AML and UML

AML is not defined from the scratch. UML was used as the underlying foundation of AML, for its language definition principles (metamodel, semantics and notation), and extension mechanisms.
AML is defined as conservative extension of UML as much as possible. That means it retains the standard UML semantics in unaltered form where possible.

AML introduces extensions to UML notation and metamodel. It also extends OCL by adding operators for modal family logics.
Some of the UML metamodel extension points are [Cervenka 2004]:

- Type (extended by EntityType…)
- NamedElement (extended by MentalState…)
- Class (extended by MentalClass…)
- Property (extended by RoleProperty, MentalProperty, ServicedProperty…)
- Association (extended by PlayAssociation, MentalAssociation…)

Thanks to the proximity of AML and UML, AML can be straightforwardly presented in UML using profile for AML. The definition of UML 2.0 and 1.4 profiles for AML can be found in the [Cervenka et al 2004] in chapter AML as UML Profile.

7.3 Description of AML Elements

In this section I will describe the elements that were used for code generation and show few notation examples. I will be introducing the elements according to their membership in respective packages.
The highest level packages in AML are Architecture, Behaviors and Mental. Code generation is using only packages Architecture and Behaviors. Package Mental contains elements like mental states, goals and beliefs which are not required to be formally specified and thus are not suitable for automated generation.
The descriptions of elements were extracted from AML Language Specification [Cervenka et al 2004] semantics and glossary. For each element I also indicate what stereotype is used in the UML 2.0 profile for AML.
At the end of this chapter I show few examples that illustrate the notation usage of the AML.

7.3.1 Architecture Package

This package is used when trying to capture architectural features of the system.
Elements from each subpackage can be used to capture one aspect of the Agent architecture.
Package Agents contains only metaclass AgentType that is used to represent agent in multi-agent systems.
SocialAspects represent structural characteristics of socialized entities – i.e. architecture of the entities that can take part in social relations.
MASDeployment is used to model deployment and residing points of agents in multi-agent systems.
Metaclasses from Ontologies package are used to represent ontology concepts and their structuring.

The code generator uses these elements from the package Architecture:

AgentType

Stereotype: agent

AgentType is a metaclass used to model a type of agents in MAS. It represents selfcontained entity that is capable of autonomous behavior in its environment i.e. entity that has control of its own behavior, and act upon its environment according to the processing of (reasoning on)

perceptions of that environment, interactions and/or its mental attitudes. There are no other entities that directly control the behavior of AgentType entity.

ResourceType

Stereotype: resource

ResourceType is a metaclass used to model types of resources contained within the system. It's able to own capabilities, observe and effect its environment, participate in social interactions, provide and use services and play roles.

EntityRoleType

Stereotype: entity role

EntityRoleType is metaclass that represents a coherent set of features, behaviors, participation in interactions. It's also able to own capabilities, observe and effect its environment and participate in social interactions

RoleProperty

Stereotype: role

RoleProperty is a specialized Property (from UML) used to specify that an instance of its owner can play one or several entity roles.

PlayAssociation

Stereotype: play

PlayAssociation is a specialized Association (from UML) used to specify RoleProperty in the form of an association end.

AgentExecutionEnvironment

Stereotype: agent execution environment

AgentExecutionEnvironment is a specialized ExecutionEnvironment (from UML) used to model types of execution environments of multi-agent system. It's able to own capabilities, observe and effect its environment, provide and use services. AgentExecutionEnvironment thus provides the physical infrastructure in which MAS entities can run.

HostingProperty

Stereotype: hosting

HostingProperty is a specialized ServicedProperty (p. 272) used to specify what EntityTypes can be hosted by what AgentExecutionEnvironments.

HostingAssociation

Stereotype: hosting

HostingAssociation is a specialized Association (from UML) used to specify HostingProperty in the form of an association end.

Ontology

Stereotype: ontology

Ontology is a specialized Package (from UML) used to specify a single ontology.

OntologyClass

Stereotype: oclass
OntologyClass is a specialized Class (from UML) used to represent an ontology class (called also ontology concept or frame).

OntologyUtility
Stereotype: outility
OntologyUtility is a specialized Class (from UML) used to cluster global ontology constants, ontology variables, and ontology functions/actions/predicates modeled as owned features.

7.3.2 Behaviors Package

This package deals with behavior decomposition from multiple aspects. Either decomposition by capabilities (BehaviorDecomposition), communication patterns (Communicative Interactions), groups of described capabilities (Services) or interactions with the external world of the agent

The code generator uses these elements from the package Behaviors:

BehaviorFragment
Stereotype: behavior fragment
BehaviorFragment is metaclass used to model coherent and reusable fragments of behavior and related structural and behavioral features, and to decompose complex behaviors into simpler and (possibly) concurrently executable fragments. It's able to own capabilities, observe and effect its environment, provide and use services.

CommunicationMessage
Stereotype: communication
CommunicationMessage is used to model communicative acts of speech act based communication in the context of 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.

InteractionProtocol
Stereotype: IP
InteractionProtocol is an Interaction template used to model reusable templates of CommunicativeInteractions. It is used to model parameterized model speech act based communications.

ServiceSpecification
Stereotype: service specification
ServiceSpecification is specialize BehavioredClassifier (from UML) used to specify services.

ServiceProtocol
Stereotype: SP

ServiceProtocol is a specialized InteractionProtocol used to specify how the functionality of a service can be accessed.

ServicedProperty

Stereotype: serviced

ServicedProperty is a specialized Property (from UML), used to model attributes that can provide or use services. It determines what services are provided and used by the entities when occur as attribute values of some objects.

ServicedPort

Stereotype: serviced

ServicedPort is a specialized Port (from UML) and ServicedElement that specifies a distinct interaction point between the owner and other ServicedElements in the model. The nature of the interactions that may occur over a ServicedPort can, in addition to required and provided interfaces, be specified also in terms of required and provided services (p. 271), particularly by associated provided and/or required ServiceSpecifications.

ServiceProvision

Stereotype: provides

ServiceProvision is a specialized Realization dependency (from UML) between a ServiceSpecification and a ServicedElement, used to specify that the ServicedElement provides the service specified by the related ServiceSpecification.

ServiceUsage

Stereotype: uses

ServiceUsage is a specialized Usage dependency (from UML) between a ServiceSpecification and a ServicedElement, used to specify that the ServicedElement uses or requires (can request) the service specified by the related ServiceSpecification.

7.3.3 Examples

Example 1

Example on Figure 3 shows a scenario with agent that implements network security management tasks.

Its behavior is decomposed to two behavior fragments – CollectLogs and EvaluateSecurityInformation. These behavior fragments could be used by other agents as well.

NetworkSecurityAgent can also play role of AccessController. AccessController forms a coherent set of functionality that takes care of assigning bandwidth provided by Bandwidth resource agent to the users.

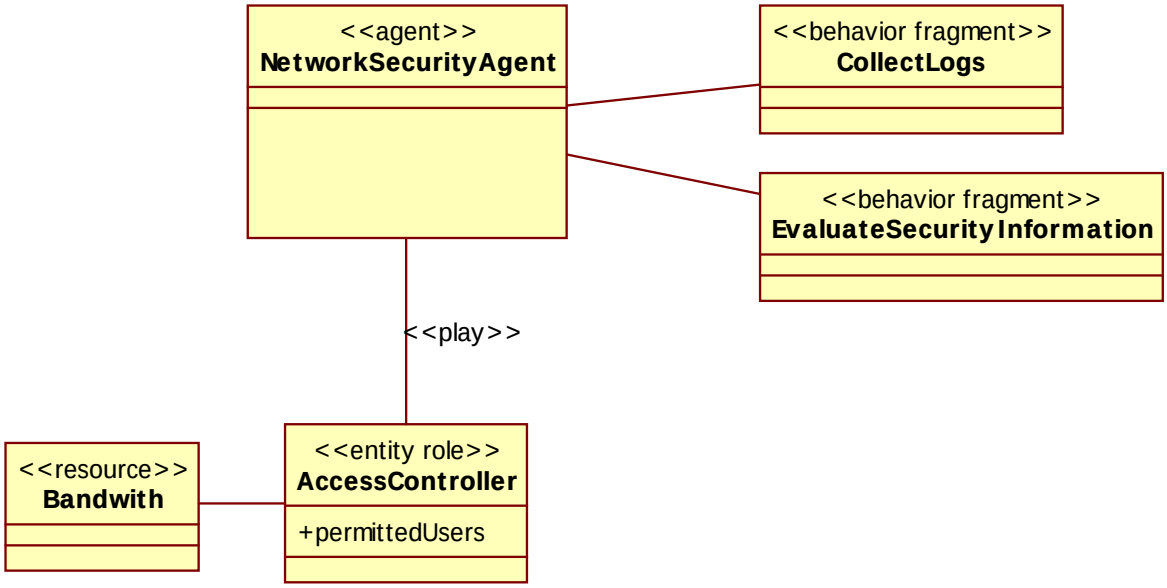

Figure 3: Example 1

Example 2

Figure 4 is further elaborating the previous example. It is showing a situation when port of NetworkSecurityAgent (userNameResolver) is using serviced specified by UserNamingService. This service is implemented by DomainController agent.

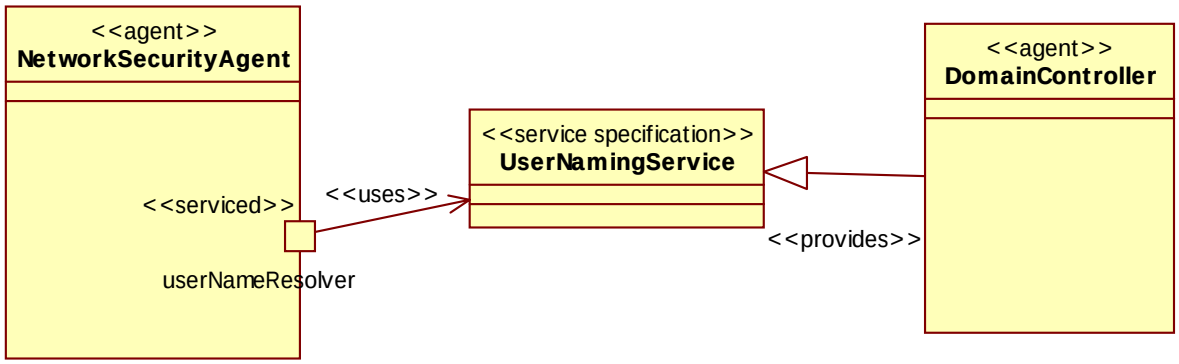

Figure 4: Example 2

Example 3

This example (Figure 5) shows communicative interaction between agents. User initiates the communication. AccessController after receiving message retrieves user name of the requesting user and informs the user whether the access was granted or not. This scenario could be used as interaction protocol if it was marked with stereotype IP.

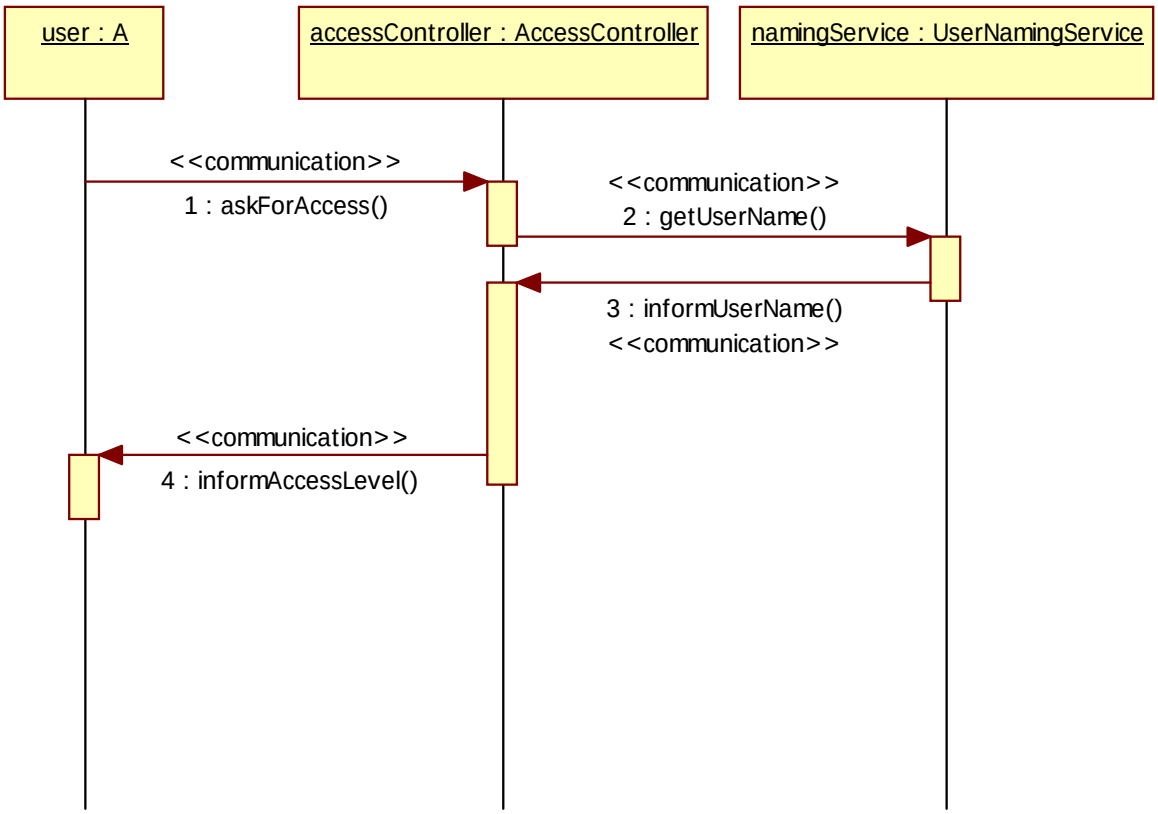

Figure 5: Example 3

8 Overview of JADE Platform

In this chapter I will provide overview of Jade agent platform in the scope necessary for understanding and usage of the code generator.
Section Runtime Environment will give an overview of JADE architecture from the perspective of multiple running JADE platforms. Next section – Programming model will provide list of the main programming concepts that are used in the target code and their brief description.
For more complex introduction into the JADE programming refer to the beginners programming guide [Caire 2003]. For detailed specifications please refer to the JADE Programmer's guide [Bellifemine et al 2005] or JADE API documentation [TILaB].

8.1 Introduction

JADE is in [Caire 2003] defined as "middleware that facilitates the development of multi-agent systems".
It consists of:

- Runtime – a platform that provides environment for the agents and facilitates basic tasks
- Library of Classes – which have to be used (inherited or instantiated) in order to utilize capabilities of the runtime
- Graphical tools – for monitoring and administrating the platform

8.2 Runtime Environment

Runtime environment is a platform that provides environment for agents to live in. In JADE it is designed to provide a transparent network platform for agent execution. Network transparency is achieved using the concept of containers. Each running runtime environment is called a container. This provides basic services and before all ensures connectivity with other containers. One of the containers is marked as "Main container" which means all other containers are registering with this container. All containers registered with the main container form a platform. Agents within a platform are able to send messages to each other without explicitly knowing the receiver agent's location. If new main container is started it starts forming a new platform.
In main container there are two special services – Agent Management System and Directory Facilitator. They are implemented in form of special agents that are started automatically when container is started.
Agent Management System (AMS) is the main infrastructure service – among other services it maintains a naming service ensuring that each agent has a unique name and helps in administrating the containers (e.g. by enabling to create/kill agent on certain container).
Directory Facilitator (DF) is the main directory (Yellow Pages) service – it facilitates searching for a specified agent. DF agent keeps list of registered agents and their descriptions. It provides interface for searching through these descriptions as service.
One example of running JADE runtime environment can be seen in Figure 6

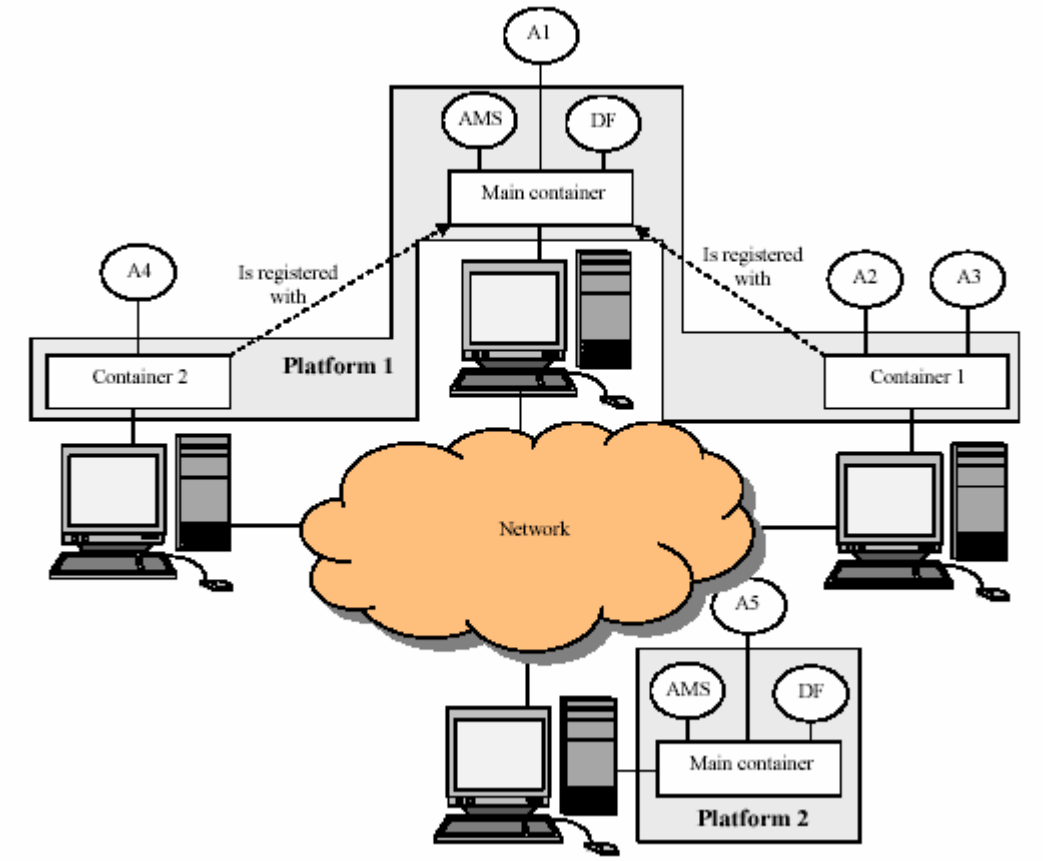

Figure 6: Example of running JADE platform (from [Caire 2003])

8.3 Programming Model

By programming model in this section I mean a set of classes, their lifecycles and design idioms used to implement applications in agent platform.
Three main concepts used in the JADE programming model are Agent, Behavior and Message. I will introduce them in the following sections.

8.3.1 Agent

Agent (jade.core.Agent) is a base class for implementing any agent that should live in the JADE environment and make use of its services. Agent class implements some of the basic features of agent – identity, autonomy, repeated execution, asynchronous messaging and mobility [Lucny 2004] In the JADE platform these properties are implemented as follows.

- Identity
  - each Agent class holds a unique identifier (Agent ID) and each running agent is unique instance of Agent class
- Autonomy
  - each class runs in its own thread.
  - Communication with agent is performed via asynchronous messaging that allows agent to be in better control of its behavior i.e. agent can decide to ignore message.
- Repeated execution

    - available behaviors (functionality fragments) are repeatedly selected and executed until they declare themselves finished
- Asynchronous messaging
    - agents can construct messages to be sent to other agents (also on different platforms). Messages are sent in a non-blocking way (agent does not wait until response from the addressee is received). Received messages are stored in the message queue that can be examined as deemed fit. As result agent can decide itself when to respond to certain message
- Mobility
    - agents can move from one container to another either by calling methods in the mobility API or sending message with special ontology to the AMS

Lifecycle of agent thread can be seen on the Figure 7.

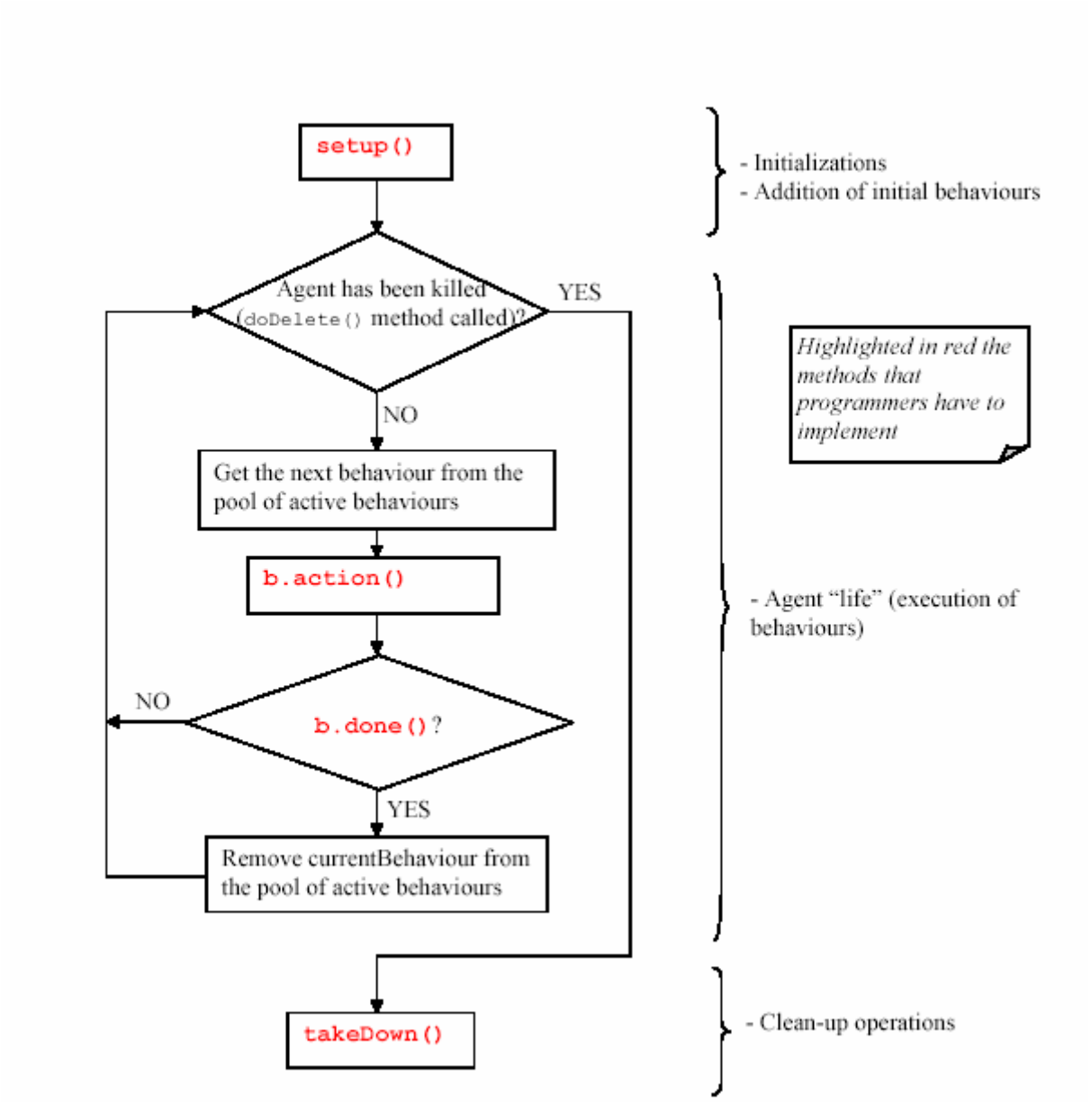

Figure 7: Agent lifecycle (from [Caire 2003])

8.3.2 Behavior

Agents in order to perform more complex tasks should be multitasking i.e. agent should be able to execute multiple tasks concurrently. This idea is abstracted into the concept of behavior (jade.core.behaviours.Behaviour). It usually implements coherent set of functionality (e.g. handling certain messages, communicate via interaction protocol etc.).

Agent can own multiple behaviors but at each time there is only one behavior active. The behavior scheduler (transparent to the programmer) performs round-robin non-preemptive scheduling of behaviors. After being selected for execution, the action() method of behavior is called. It should handle all the necessary steps of the process. Note that until method action() is finished no other behavior gets to be executed. Behavior is scheduled for execution executed as

long as method done() returns false. After done() returns true, behavior is removed from the pool of active behaviors where it can be explicitly returned if necessary.

8.3.3 ACLMessage/MessageTemplate

Sending messages is the main mean of communication between agents. Concept of message is implemented in the jade.lang.acl.ACLMessage class. It represents message in Agent Communication Language (ACL) that can be exchange between agents. If agent wants to send a message it should create new instance of ACLMessage, fill its parameters and then call method Agent.send().

ACLMessage can carry, among others, following information (see exact names in the API reference):

- sender – sending agent
- receiver – receiving agent (message can be sent to multiple receivers)
- reply-to – agent that a reply should be sent to
- performative (communicative act) – indicates what is the purpose of the message. One of the constants for FIPA performative (PROPOSE, REQUEST, REFUSE…)
- content – payload of the message in form of string or java object
- in-reply-to – identifies the message that current message is responding to
- language – language of the message
- ontology – name of ontology that can be used to decode the message
- conversation ID – identification of the conversation of which current message is part of

After receiving messages are put into the agent message queue. If the agent needs to find specific message in the message queue it should use MessageTemplate.

MessageTemplate (jade.lang.acl.MessageTemplate) is class that lets user to set parameters by which he or she wants to match the messages in message queue. MessageTemplate contains set of static methods that allow matching most attributes of the ACLMessage and also combining multiple criteria. This process is usually performed in following steps:

- create instance of MessageTemplate
- define values of attributes to be matched using MessageTemplate static methods (e.g. MatchConversationID, MatchPerformative, MatchInReplyTo…)
- call Agent.receive() with message template as parameter

9 AML to JADE Mapping

This chapter will define a mapping between the Agent Modeling Language (AML) and JADE platform. The chapter is organized into sections, each describing mapping of one package from AML. The description for each pattern is structured as follows:

- Mapping definition – contains description on the mapping from the source pattern in the AML to the destination Java/JADE code.
- Constraints – shows the constraints that have to be adhered in order the code generator functions properly.
- Rationale – this section discusses on the meaning of the transformation and its correctness according to the AML specification and JADE programming model.

9.1 Mapping Definitions

9.1.1 Agent

Mapping

Agent is mapped into class jade.core.Agent with following properties:
Attributes of AML Agent are directly transferred to JADE Agent
Operations of AML Agent are directly transferred to JADE Agent
Operations of AML EntityRole that is associated to the AML Agent are directly transferred to JADE Agent.
Any behavior that is generated from the object of type AML Agent in sequence diagram is added to the Agent. Resulting JADE Behaviour is added to the JADE Agent in the setup() procedure.

Constraints

Play association has to be between AgentType and RoleType

Rationale

Attributes and operations in JADE provide similar semantics as those in the AML so they can be directly transferred into the JADE implementation.
RoleType defines the "coherent set of features, behaviours, participation in interactions and services" so it is treated as AML Agent and all behaviours are added into the JADE agent when agent is playing role

9.1.2 RoleType

Mapping

Following mapping mechanism is used in implementing RoleType element:

- The amlextensions.Role class is implemented. Role class handles registration of behaviours that are connected with the role and their registering or deregistering with agents.

- Any RoleType element inherits from this class or contains this class as mix-in functionality class
- Behaviours that are defined by RoleType element are added to the inherited Role class in the constructor
- Agent contains collection Agent.playedRoles – a hashtable of Roles that can be played by this agent.
- According to situation agent can call playRole(roleName) procedure to register behaviours corresponding to role called roleName or call disposeRole(roleName) to deregister behaviours corresponding to role called roleName. roleName has to be conained in the Agent.playedRoles collection.

Role can appear in following contexts:

- Class diagram – class marked with stereotype <<entity role>>
  - Results into generation of class inherited from amlextensions.Role with attributes and operations as indicated in the AML element
- Sequence diagram – instance of EntityRoleType.
  - Results into generation of message handler that can send/respond to messages in the interaction
  - Generated message handler (specialized Behaviour) is registered as part of the EntityRoleType.
  - For details on behaviour generation see BehaviourFragment

Rationale

EntityRoleType represents “coherent set of features, behaviors, participation in interactions, and services” [Cervenka et. al 2004].
The relation to the agent can be set in the class diagram using PlayAssociation. Since the agent can play different role at different time they can be dynamically changed using playRole and disposeRole methods.
Role is composed of jade.core.behaviours.Behavior because they are the main source of activity in the JADE platform (see chapter 8 Overview of JADE). All interaction among agents are performed using behaviours, so they are natural choice for handling messages in interactions that EntityRoleType takes part in. For rationale on MessageHandler generation please refer to MessageHandler section in this chapter.

9.1.3 PlayAssociation

Mapping

Play association is not expressed explicitly in the resulting Java/JADE code.
It is used to indicate which roles are registered with Agent. Each role that is connected to the Agent using PlayAssociation is added to the Agent.playedRoles collection in the Agent.startup() method.

Constraints

Play association can be used only between AgentType and EntityRoleType elements.

Rationale

PlayAssociation indicates which roles can be played by agent. This is achieved in the JADE implementation by registering all linked roles to the agent which can invoke them when needed.

9.1.4 BehaviorFragment

Mapping

BehaviorFragment is mapped into a jade.core.behaviours.Behavior class. If it is linked to the Agent or Role using association it is registered as behavior for the corresponding element.

Rationale

BehaviorFragment serves as a decomposition of the agent behavior. As such it is generated into the Behaviour class that is registered with agent. It provides the agent added functionality as defined by BehaviorFragment.

9.1.5 AgentType Communication

Mapping

The result mapping of agent participating in the communicative interaction is a specialized BehaviorFragment that is used solely for handling messages.

For each instance of AgentType or EntityRoleType that takes part in the communicative interaction, a separate BehaviorFragment is generated. It handles messages in following way:

- Behaviour contains attribute stateNumber that captures ID of the current state
- Action method examines the stateNumber and decides which state to put into operation
- Each state handles one message – generator provides template for handling messages by creating a message template, call to send or receive message and blocking of behaviour if message is not present yet.
- If the message is sent by instance in the interaction – message with proper parameters (message name, receiver…) is constructed in one of the states of the behaviour. Also a setReplyWith is called in order to set identifier that will be used to retrieve response to this message. As preemptive step a messageTemplate is constructed that can be used to pick correct response message from the message queue.
- If the message is received by instance in the interaction – message template is adjusted to suit the situation. Then attempt is made to retrieve the message. If message is not present the behaviour is blocked until another message arrives.

Rationale

Message handling behavior (jade.core.behaviours) in this context is used as decomposition of the message handling capabilities of the agent. Each instance is generated into separate message handler to ensure consistency of protocol that is performed by participant.

9.1.6 Communicative Interaction

Mapping

Interaction is enclosing element for message handling behaviors. It is not explicitly generated in the output code.

Constraints

All participant types in the communicative interaction have to be agents.

Rationale

Communicative Interaction is used as logical unit to group message handling behaviors in one interaction. There is currently no justification for it to appear in the output code explicitly.

9.1.7 InteractionProtocol

Mapping

Interaction protocol is parameterized Interaction where each of the message handling behaviors can be assigned a new classifier.
InteractionProtocol is generated as factory class from which user can obtain Behaviours with bound parameters (instance classifier and message name/payload).
If InteractionProtocol parameters are bound in the diagram, behaviours are registered with their respective agents/roles.

Constraints

Compared to AML specification only lifeline classifiers and message names can be parameterized.

Rationale

According to AML specification parameters of InteractionProtocol can be bound on various places in the model. Due to limitations of StarUML the parameter binding of InteractionProtocol is not supported in the model. However in the code user can utilize the functionality of factory class to obtain parameterized behaviors. In this place the code generation is not direct transformation but provides extension to JADE that can be utilized by user.

9.1.8 AgentExecutionEnvironment

Mapping

AgentExecutionEnvironment is mapped into the directory that contains file residing on the agent platform.
For each type of the hosting properties or hosting associations a file containing code is generated and put into folder created for AgentExecutionEnvironment.

Rationale

AML is not strict on the detailed specification of AgentExecutionEnvironment. It should provide physical infrastructure in which MAS entities can run. Mapping into the directories containing

hosted artefact does not directly affect execution environment of agents but prepares the files necessary for deployment on target platform.

9.1.9 HostingProperty

Mapping

HostingProperty type is added to the collection of deployed artifacts by AgentExecutionEnvironment.

Rationale

HostingProperty specifies that EntityType can be hosted in AgentExecutionEnvironment. Proposed mapping is not directly enforcing this but it is facilitating deployment of HostingProperty type to the target platform.

9.1.10 HostingAssociation

Mapping

hostingMemberEnd type of HostingAssociation is added to the collection of deployed artifacts by AgentExecutionEnvironment.

Rationale

Semantics of HostingAssociation is similar to the semantics of HostingProperty. Therefore they share the same rationale.

9.1.11 ServiceSpecification

Mapping

Following mapping mechanism is used in generating ServiceSpecification element:

- The amlextensions.Service class is implemented. Service class handles following tasks:
  - Registering/deregistering service implementation by specified agent
  - Retrieve agent identifiers that implement service
  - Provide behaviour (message handler) that can communicate with the service on the side of the service client.
- Any ServiceSpecification element inherits from amlextensions.Service class or has instance of this class as mix-in functionality class.
- Behaviours that are defined by ServiceSpecification element are added to the inherited Service class in the constructor

Rationale

Result of ServiceSpecification generation serves as helper class that facilitates operations with services. This is contribution compared to the manual programming because handling service registration, deregistration and providing agents retrieval requires extra amounts of code.

9.1.12 ServicedElement

Mapping

ServicedElement is abstract class and as such is not mapped into any concept in JADE. See its concrete classes for mapping details.

9.1.13 ServicedProperty

Mapping

Used service specification is added to the used services collection of serviced property type.
Provided service specification is added to the provided services collection of serviced property type.

Rationale

Serviced property indicates that its type is able to provide or use services. In order the user can easily access service specification they are kept in the collections which make service specifications readily available. They can be for instance used as factory classes to create behaviors necessary to communicate with service provider.

9.1.14 ServicedPort

Mapping

Used service is added to the used services collection of serviced port type.
Provided service is added to the provided services collection of serviced port type.

Rationale

ServicedPort is a special kind of ServicedProperty. Therefore the same rationale applies.

9.1.15 ServiceProvision

Mapping

Adds provided service specification to the ServicedElement type (e.g. port type, AgentType etc.).

Constraints

Connects ServicedElement (ServicedPort, ServicedProperty, AgentType, RoleType…) and ServiceSpecification

Rationale

Along with the specification of the ServiceProvision it indicates “ServicedElement provides the service specified by the related ServiceSpecification”. By adding ServiceSpecification to the provided service of ServicedElement type it allows the behaviours that implement the service to be generated.

9.1.16 ServiceUsage

Mapping

Adds used service service specification to the ServicedElement type (e.g. port type, AgentType etc).

Constraints

Connects ServicedElement (ServicedPort, ServicedProperty, AgentType, RoleType…) and ServiceSpecification

Rationale

Along with the specification of the ServiceUsage it indicates “ServicedElement uses or requires (can request) the service specified by the related ServiceSpecification”. By adding ServiceSpecification to the used services of ServicedElement type it allows the behaviours necessary for service usage to be generated and added to the owning jade.Agent.

10 Implementation Description

Parts of this work are two executable artifacts – UML profile for AML implemented in StarUml and AML-JADE code generator. In first two sections of this chapter I will introduce their architecture and design. The third section will discuss the algorithm used for the pattern matching and its complexity.

10.1 Architecture

Architectural considerations were discussed in the chapter 4. In this section I will just summarize the results to recall architecture of code generator before the description of the design.

As concluded in the chapter 4 the most suitable architecture for the code generator that will be implemented is as follows. I will use the pipe architecture as in conventional compiler structure. The process of code generation will have two phases: transformation from platform independent model to platform specific model as first phase and platform specific model to target code as second phase.

Source for the code generation will be an AML model implemented as UML profile in StarUML. From the point of view of MDA the AML model can be viewed as platform independent model.

The main control logic will reside in the StarUML add-in which will ensure seamless integration with the UML metamodel implementation available in this CASE tool. The add-in (frontend) will be transforming the AML model to internal representation of platform specific model. Transformation will be based on the procedural pattern definitions. It means that patterns are not defined as model fragments but as procedure that describe how respective pattern should be recognized.

Note that there is no explicit pure JADE model. It could constitute one more step in the process between PSM and target code. This step was omitted for following reasons. JADE elements are defined as Java classes and as such there is no structural difference between JADE elements and plain Java classes. In addition AML concepts themselves are close enough to JADE concepts so that PSM model is already straightforward to transform to JADE. Additional step would not add any clarity or value to the process.

The backend generator will be implemented as independent library that transforms platform specific model to the target code according to procedural pattern definitions similar to those of frontend generator. Despite backend generator is independent library it will be controlled by the application running as the StarUML add-in.

10.2 Design and Implementation

The above mentioned architecture was implemented using the following model.

The whole generator is wrapped in the AMLGenerator class. It handles setting up the generation and provides simple interface to obtain target code. AMLGenerator is composed of three main parts – ModelBrowser, PatternRecognitor and BackendGenerator. Each of them supports one step in the transformation process.

ModelBrowser handles retrieving elements from the UML model. It traverses the model in breadth first order and at each turn it adds children of currently visited element as next targets. Then it returns currently visited element. ModelBrowser also keeps list of visited elements to make sure that each element is visited only once.

PatternRecognitor handles transformation between PIM (AML model) and PSM (mixed AML/JADE model). Input for PatternRecognitor are elements that are returned by ModelBrowser.
Pattern recognition procedures (procedural templates) are implemented in the PatternDefintion objects owned by PatternRecognitor. Each pattern definition can handle recognition of one or more types of patterns and every pattern definition receives each element coming from the ModelBrowser.
Functionality of PatternRecognitor can be described in pseudo-code as follows:

```
function Recognize(Element)
begin
  recognizedPatterns={}
  for each pattern in patternDefinitions
      recognizedPatterns = recognizedPatterns U pattern.Recognize(Element)
  return recognizedPatterns
end
```

Separation of pattern definitions from pattern recognitor provides a way how to customize which patterns will recognized without modifying PatternRecognitor.
Other feature of pattern recognitor is to retrieve recognized patterns according to UML element which they were generated from. This function is used when PatternDefinition object needs information about the structures that it is not able to recognize.
Typical pattern definition checks the stereotype of the examined element, creates the new pattern instance, adds structural features (attribute and method definitions, received messages…) to the pattern instance. Additionally it may add child patterns (e.g. played roles for agent).
If pattern definition handles association, dependency or other form of link it will typically retrieve patterns of its ends and modify these patterns so that they reflect the relationship indicated by link. For example play association will modify agent pattern so that it will add role at the other association end to the played roles collection.
BackendGenerator implements the transformation from PSM to target code. BackendGenerator iterates through the recognized patterns and according to pattern type generates text output.
For most patterns backend generator will generate a Java class (as instance of JavaClass class). Most of the generated classes inherit from jade.core.Agent, jade.core.behaviours.Behaviour or jade.lang.acl.ACLMessage which are basic classes to work with in JADE. Backend generator then adds attributes and methods, adjusts constructor to fill class collections (e.g. add behaviors for agent) or initialize instance attributes. Depending on the pattern type backend generator may also add other methods to the class (e.g. Behaviour.action() methods to specify behaviour execution process). After class is constructed it is output into text file. For description how AML elements are mapped into JADE classes refer to the chapter AML-JADE mapping.
Each of the components resides in separate package with very narrow interfaces so they can be replaced by other package of the same functionality if needed.
The public interface point of ModelBrowser is iterator-like interface that provides information if there is element available (HasNext()) or retrieves this element (GetNext()).
PatternRecognitor main method is RecognizePattern() that takes UML element as attribute and returns the pattern recognized by one of the pattern definitions.
BackendGenerator main functionality is encapsulated in the GenerateFS() method that returns a class representing root directory of the generated code.

10.3 Algorithms and Complexity

UML code generation is dealt with in various works that show how it can be often formalized as graph pattern matching problem. For example see [AGRAWAL]. In this setting the UML model is labeled graph and graph patterns define the fragments from which code can be generated. Pattern matching is then performed in order to find these patterns in the source UML model.
AML to JADE code generation can be formalized in similar manner. However, thanks to the resembling structure of AML and JADE object model, the problem is simpler (in terms of pattern matching) than general UML code generation.
Let AML model be represented by labeled graph G. We can obtain a graph G by applying following transformation rules:

- Each AML element is expressed as a vertex of a graph,
- Type of element (class, interaction…) is expressed as label of the vertex,
- Structural features (attribute, operation) of elements are represented as labeled vertexes connected to the parent element by edge,
- Each structural feature or instance is linked to the classifier by specially labeled edge. The connected vertex indicates the type or classifier of the structural feature
- Parameters of operation are handled as structural features of operation,
- Let there be elements A and B and link L that connects them. Let a and b be the vertexes that correspond to these elements in the graph G generated using previous rules. Then graph G will contain edge (a,b) with label indicating the type of the link L.
- Labels contain all the necessary information about the AML elements - e.g. name, type, parameter information, visibility…

Figure 8 shows the transformation of simple AML diagram to the labeled graph according to the defined rules.

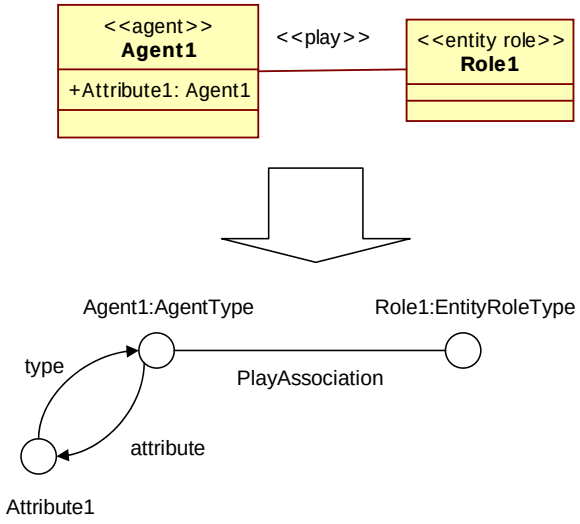

Figure 8: Transformation of AML to labeled graph

Patterns for matching are defined as any AML model fragment. They can be converted to the graph representation same as the original AML model.
As shown in the chapter 9 (AML to JADE Mapping) AML and JADE structures are similar and so the patterns needed for transformation are very simple. They usually comprise only one

element. Typical property of the patterns is that they usually describe only small surrounding of the element and more complicated structure is described by composing patterns together.
Typical pattern structure is element (e.g. Class) with some specifying relations to other elements (i.e. – decision if pattern matches often depends only on the label of the vertex, other vertices can only refine the information about this element).
If the graph and patterns had general structure, the pattern matching would be a NP complete problem. The naïve approach to graph pattern matching generates all possibilities of mappings from the pattern nodes to the target node and checks if the matching is correct. It performs $O(m^n)$ tests of matching nodes, where n is number of nodes in the pattern and m is number of patterns in target graph [Valiente].
In our special case the number of nodes in the pattern is restricted to O(1). That makes even usage of the naïve algorithm for pattern matching in the code generator feasible because it runs in polynomial time.

The modified naïve pattern matching algorithm used in the code generator:

- Get next node from source graph (e.g. breadth first search)
- Test all pattern definitions if they are feasible to start in this node (root node) (e.g. if they contain correct label - classifier or stereotype)
    - Examine all feasible neighboring elements (connected with correct link (edge label) and having proper type and stereotype (vertex label)). Recognize their patterns or retrieve their recognized pattern from the recognized pattern collection.
    - Compose all details/sub-patterns found in previous step into pattern recognized from the root vertex
- Add tested nodes and nodes used when looking ahead while pattern recognition to visited objects
- Continue with next node

In order to verify that the algorithm runs in polynomial time I will perform the worst case time complexity analysis.
In the step 1 of the algorithm every node of the graph is visited once – n (number of vertices)
For each pattern definition (size k) – try to match the subgraph induced by the root vertex (it can include matching another patterns which act as parts of the examined pattern but they don't have to be considered because they would be processed when their root vertex would be chosen).
Only action that has to be considered here is searching for the pattern whether it was already recognized due to activity in some other node. Because there is approximately 1:1 mapping between AML and JADE patterns there can be at most $O(n)$ patterns in the recognized patterns collection. So each time search is performed it takes time $O(n)$.
Let m be maximum grade (number of neighbors) of vertex in graph. Then for each neighbor (out of m) test the feasibility for all pattern definition (k feasibility tests per neighboring node) => $m*k$ feasibility tests per visited node. This test is performed for each node but it is ensured that each pattern is tested only once so there is no duplication.
The final worst case complexity adds up to: $n*l*m*k$ where n is number of vertices in graph, l is maximum number of recognized patterns and can be approximated as n, m is grade of graph and k is maximum number of pattern vertices.
This theoretical time complexity is in accordance with the expected complexity and fitting for this application. Also in practical tests the performance seems satisfactory.

11 Conclusion and Future Work

The main objective of the work was to define mapping between AML and JADE agent platform and implement it in code generator. The executable output of this work is a proof-of-concept application that is able to generate code skeletons for the JADE platform. Second artefact produced along with this work is implementation of UML profile for StarUML.

Mapping between AML and JADE was performed based on the semantics defined in the AML specification and documentation of the JADE classes.

Most of the metaclasses that are used to model static structures of the agent were mapped to JADE concepts.

Among the main elements that were used for code generation belong those of type: AgentType, EntityRole, BehaviorFragment, ServiceSpecification etc. Mapping of these elements was more or less straightforward thanks to the proximity of concepts in AML and JADE. One of the complications that I had to deal with was incomplete UML metamodel implementation provided by StarUML.

Easy mapping shows that AML is defined in a way that makes it easy to transfer models to the concrete implementation. This feature gives AML advantage in terms of supportability by tools. From the JADE perspective the straightforward mapping also shows that programming model used in JADE is close to the multi-agent system concepts. It means that JADE as agent platform brings programming of multi-agent application to higher level of abstraction. These results are favourable for both AML and JADE.

However the mapping was not defined for some of the elements for which precise generation a formal operational semantics would be required. As mentioned in the AML specification the operational semantics is outside the scope of AML. There is possibility of future work that would choose one or more operational semantics and code generation from AML according to their specifications.

Another part of the AML that was not implemented form elements for describing mental states of agent. Since JADE does not natively support the reasoning of intelligent agents these elements were left unsupported. One possibility to cope with this issue could be exploring connection of JADE and reasoning engine (e.g. JESS [!ref]) that could form to the platform that would be able to capture also this part of AML. Other platform based on JADE that is capable of reasoning is JADEX [Braubach 2006].

Generally the biggest contribution is setting the solid foundations for future works on code generation from AML and showing an example of mapping between AML and JADE agent platform. Also the work leads you through the design of the code generator from the architectural foundations, through the technological considerations to the design and implementation description.

12 Appendix 1

12.1 List of Figures

Figure 1: MDA Process from [Code Generation Network 2004] 10
Figure 2: Possibiliteis of control flow in code generator 13
Figure 3: Example 1 30
Figure 4: Example 2 30
Figure 5: Example 3 31
Figure 6: Example of running JADE platform (from [Caire 2003]) 33
Figure 7: Agent lifecycle (from [Caire 2003]) 35
Figure 8: Transformation of AML to labeled graph 46

12.2 List of Tables

Table 1: Scenario comparison 15
Table 2: CASE tool comparison 19
Table 3: Agent Platform Comparison 23

12.3 Reference list

[Agrawal]
AGRAWAL, Aditya, KARSAI, Gabor, SHI, Feng, Institute for Software Integrated Systems, Vanderbilt University: Graph Transformations on Domain-Specific Models [online]

[Aho 85]
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools, Addison Wesley Publishing Company; (October 1, 1985)

[Bellifemine et al 2005]
BELLIFEMINE Fabio, CAIRE Giovanni, TRUCCO Tiziana, Giovanni Rimassa, TILab S.p.A., 2005: JADE Programmer's Guide [online]

[Braubach 2006]
BRAUBACH, Lars, Pokahr, Alexander, Walczak, Andrzej, University of Hamburg: Jadex BDI Agent System – Features [online]
Available on internet: <http://vsis-www.informatik.uni-hamburg.de/projects/jadex/features.php>

[Bresciani 2002]
BRESCIANI, P, Perini, Anna, Giorgini, Paolo, Giunchiglia, Fausto, Mylopoulos, John et al., Kluwer Academic Publishers, 2002: Tropos: An Agent-Oriented Software Development Methodology [online]
In Autonomous Agents and Multi-Agent Sytems, 8
Available on internet: <http://www.dit.unitn.it/~pgiorgio/papers/jaamas04.pdf>

[Caire 2003]
CAIRE Giovanni, TILab S.p.A, 2003.: JADE Tutorial – JADE Programming for Beginners

[Cervenka et. al 2004]
Cervenka Radovan, TRENCANSKY Ivan, Whitestein Technologies AG, December 2004: Agent Modeling Language – Language Specification

[Cervenka 2004]
Cervenka Radovan, Whitestein Technologies AG: Agent Modeling Language – Metamodel v0.9
[Code Generation Network 2004]
Code Generation Network – MDA [online]
Available on internet: <http://www.codegeneration.net/tiki-index.php?page=MDA>

[DeLoach 1999]
DELOACH, Scott A., Department of Electrical & Computer Engineering, Department of Electrical & Computer Engineering: Multiagent Systems Engineering: A Methodology And Language for Designing Agent Systems [online]

[Eckel et al. 2003]
ECKEL, Bruce, MindView, 2003: Bruce Eckel's Mind View Inc, UML Tool Survey [online]
Available on internet: <http://www.mindview.net/WebLog/log-0041>

[Evans 2001]
EVANS, Richard, et. al, Braodcom Eireann Research Ltd., 2001: MESSAGE: Methodology for engineering systems of software agents. Methodology for agent-oriented software engineering. [online]
Available on internet:
<http://www.eurescom.de/~pub-deliverables/P900series/P907/TI1/p907ti1.pdf>

[FIPA]
FIPA (The Foundation for Intelligent Physical Agents) web page [online]
Available on internet: <http://www.fipa.org>

[Gentleware 2006]
GENTLEWARE: Poseidon Project Home Page [online]
Available on internet: <http://gentleware.com/index.php?id=products>

[GODFREY 2006]
GODFREY Michael W., School of Computer Science, University of Waterloo, 2006: My Little UML (Tools) Page [online]
Available on internet: <http://www.uwaterloo.ca/~migod/uml>

[Haywood 2004]
HAYWOOD Dan, May 2004: MDA Nice idea. Shame About the… [online]
Available on internet: <http://www.theserverside.com/tt/articles/article.tss?l=MDA_Haywood>

[IBM 2006]
IBM: Rational Rose home page [online]
Available on internet: < http://www-306.ibm.com/software/awdtools/developer/datamodeler/ >

[IKT++ 2006]
IKT++: Grasshopper project homepage [online]
Available on internet: <http://www.grasshopper.de>

[JACK 2006]
AGENT ORIENTED SOFTWARE GROUP, 2006: JACK Intelligent Agents, software agent system (project home page) [online]
Available on internet: <http://www.agent-software.com/shared/home/>

[Lucny 2004]
LUCNY, Andrej, Microstep-MIS, 2004: Multiagentové systémy (lecture slides)

[MARCHAL 2004]

MARCHAL, Benoît, IBM, May 2004: Working XML: UML, XMI, and code generation, Part 2 [online]
Available on internet: <http://www-128.ibm.com/developerworks/library/x-wxxm24/#code1>

[Novosoft 2002]
NOVOSOFT, 2002: NSUML (Project Homepage) [online]
Available on internet: <http://nsuml.sourceforge.net>

[OMG 2002]
OMG (Object Management Group): Request for Proposal: MOF 2.0 Query/Views/Transformations RFP [online]
Available on internet: <http://www.omg.org/docs/ad/02-04-10.pdf>

[OMG 2005]
OMG (Object Management Group), 2005: UML 2.0 Superstructure Specification [online]
Available on internet: <http:// www.omg.org/cgi-bin/doc?ptc/2003-08-02>

[OMG 2006a]
OMG (Object Management Group): home page [online]
Available on internet: <http:// www.omg.org>

[OMG 2006b]
OMG (Object Management Group): Model Driven Architecture home page [online]
Available on internet: <http:// www.omg.org/mda>

[OMG 2006c]
OMG (Object Management Group): Unified Modeling Language [online]
Available on internet: <http:// www.omg.org/uml>

[Petrie]
PETRIE, Charles J. Agent-Based Engineering, the Web, and Intelligence [online]
Available on internet: <http://cdr.stanford.edu/NextLink/Expert.html>

[QVT 2003]
QVT Partners: QVT: The high level scope [online]
Available on internet: < http://qvtp.org/downloads/qvtscope.pdf >

[SPARX 2006]
SPARX Systems: Enterprise Architect home page [online]
Available on internet: <http://www.sparxsystems.com.au>

[StarUML 2006]
STARUML Development Group: StarUML Project Home Page [online]
Available on internet: <http://www.staruml.com>

[Tigris 2005]
TIGRIS: ArgoUML Project Home Page [online]
Available on internet: <http://argouml.tigris.org/>

[TILaB]
TILaB S.p.A: JADE v3.3 API [online]

[TILab 2006]
TILAB: JADE – Java Agent DEvelopment Framework (project homepage) [online]
Available on internet: <http://jade.tilab.com>

[Valiente]

VALIENTE, Gabriel, Martinez, Conrado, Universitat Bremen, Fachbereich Mathematik und Informatik: An algorithm for graph pattern-matching [online]

[Whitestein 2006]
WHITESTEIN Information Technology Group AG: Whitestein Technologies: Technology Suite [online]
Available on internet: <http://www.whitestein.com/pages/solutions/ls_ts.html>

Abstrakt

Generovanie kódu je technika pomocou ktorej sa transformuje analytický model na artefakty. V tejto diplomovej práci budem prezentovat implementáciu generátora kódu, ktorý prekladá agentové modely v AML do kódu spustitelného na agentovej platforme JADE.

Architektúra kód generátoru je založená na vzore rúra (pipe), ktorá je základom klasickej štruktúry kompilátoru. Postupnosť transformácií modelu je ovplyvnená architektúrou MDA (Model Driven Architecture – Architektúra Riadená Modelom). Podobne ako v MDA kód generátor využíva dve fázy generovania: najprv pretransformuje model nezávislý na platforme (PIM) na platformovo závislý model (PSM). V druhej fáze generuje z platformovo závislého kódu zdrojový kód pre agentovú platformu.

Aby bolo možné implementovať generátor kódu bolo nutné definovať mapovanie z jazyka AML do jazyka cieľovej agentovej platformy. Ukázalo sa, že je možné rozdeliť mapovanie na tri skupiny: priamočiare mapovanie elementov, ktoré sú svojou povahou príbuzné v AML aj platforme JADE. Sem patrí napríklad koncept agent alebo správanie. Druhá skupina sú elementy, ktoré sa kvôli obmedzeniam CASE nástroja dajú modelovať iba približne. Príkladom môže byť interakčný protokol. Tretiu skupinu tvoria elementy ktoré buď nie sú v AML definované presne alebo nie sú dostatočne podporované agentovou platformou. V tejto skupine sú napríklad mentálne stavy.

Výsledkom práce je funkčný prototyp kód generátoru ktorý je možné ďalej rozvíjať.

Výsledkom pr