Modelling MECA scenarios using Brahms and KAoS

Thesis Bachelor Project IN3405

Lennard de Rijk	Pieter Senster	Ot ten Thije
1308270	1223577	1282859

Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science

TNO Defence, Security and Safety
Department Smart Assistance

June 23, 2009

Exam committee:

Nanja Smets M.Sc.
TNO Defence, Security and Safety, Soesterberg

prof. dr. ir. Catholijn Jonker
ir. Matthijs Sepers
Delft University of Technology

Preface

This report was written in context of the final project for Bachelor students Computer Science at the Delft University of Technology, completed at TNO Defence, Security and Safety. The goal of this project is for groups of students to complete a full software engineering process. The project spanned eleven weeks, from April 14th to July 1st 2009.

In our project we combined the Brahms multi-agent programming language and the KAoS policy service to develop a prototype for a human-in-the-loop experiment. One of the highlights of the project was a presentation of this bridge to the Brahms developers at NASA AMES Research Center, where it was received enthusiastically.

We would like to thank Maarten Sierhuis for his support and input on the Brahms code of our library and simulation. We are also grateful to Ron van Hoof, who helped us solve many practical issues we encountered in Brahms. Finally we would like to thank Jurriaan van Diggelen for getting us started with KAoS and his critical reviews on our designs and presentations.

Lennard de Rijk
Pieter Senster
Ot ten Thije

Delft, June 2009

Summary

In this report we present a simulation of a MECA scenario, powered by Brahms and KAoS. To achieve this, the KAoS/Brahms bridge is developed, which allows agents in a Brahms multi-agent system to query KAoS policies. Because the development of the bridge and the simulation proceeded quickly, the simulation could be modified for use in a human-in-the-loop experiment as well.

The KAoS/Brahms bridge consists of three main components: the Brahms Policy Library, the KAoS Gateway and the Ontology Builder. The Brahms Policy Library provides a generic API that Brahms agents can use to query policy services. This library is backed by the KAoS Gateway Agent, which provides the implementation necessary to communicate with KAoS. However, to define policies about concepts in the Brahms simulation, KAoS needs an ontology describing what those concepts are. This ontology can be automatically generated from the Brahms model using the Ontology Builder.

With the KAoS/Brahms bridge at hand, the MECA project is used as context for a scenario modelled in Brahms. The scenario is scripted to contain various situations in which Brahms' work practice approach interacts with KAoS' top-down specification of rules. The outcome of these interactions directly influences the outcome of the scenario as a whole.

By including an actual human being in the simulation, the scenario can also be used to assess man-machine interaction between this human subject and the other agents. The human subject can assume the role of one of the actors, while the other actors remain simulated. By using a graphical user interface, the human is able to influence the simulation.

We conclude that Brahms and KAoS can indeed be integrated, as demonstrated by the KAoS/Brahms bridge. The bridge described in this report supports a large part of KAoS functionality, but to unlock the full expressiveness of KAoS policies more work is still needed.

A more general conclusion is that using Brahms and KAoS it is possible to implement a scenario which relies on the principles of both work practice and policies. If this combination is further extended to include human subjects, we expect new opportunities to emerge that combine the actual human experience of the subject with the benefits of simulated agents. How the results of such experiments compare to other human-in-the-loop approaches needs to be determined by future research.

Glossary

ACL See *Agent Communication Language*.

Action Instance Description Used by KAoS to represent the request of an agent for authorisation to perform an action.

Agent Communication Language A standardised format for messages sent between agents, based on speech act theory.

AID See *Action Instance Description*.

Ant A piece of software that allows the execution of build scripts. These build scripts can be used to perform a wide range of actions, from compiling code to exporting documentation.

API Application Programming Interface.

Authorisation A constraint that permits or forbids some action.

BDI See *Belief Desire Intention*.

Belief Desire Intention A software model for developing intelligent agents. Agents use the concepts of beliefs, desires and intentions to select pre-defined plans, which are then used to solve problems.

ePartner Synonym for *Mission Execution Crew Assistant*.

Extensible Stylesheet Language A family of transformation languages specifying formatting or transformation of XML files.

FIPA-ACL An Agent Communication Language defined by the Foundation for Intelligent Physical Agents.

Habitat The command base and living area of the crew on a remote planet.

HITL See *Human-in-the-loop*.

Human-in-the-loop A model that requires human interaction.

Infinite recursion See *infinite recursion*.

KAoS See *Knowledgeable Agent-oriented System*.

Knowledgeable Agent-oriented System A system for storing and enforcing policies. Better known under its acronym KAoS.

MAS See *Multi-Agent System*.

MECA See *Mission Execution Crew Assistant*.

Mission Execution Crew Assistant System that empowers the cognitive capacities of human-machine teams during planetary exploration missions in order to cope autonomously with unexpected, complex and potentially hazardous situations.

Multi-Agent System A system in which multiple autonomous agents collaborate to achieve a common goal.

Obligation A constraint that requires some action when a state- or event-based trigger occurs.

Ontology A formal representation of a set of concepts within a domain and the relationships between those concepts.

OWL See *Web Ontology Language*.

Policy An enforceable, well specified constraint on the performance of a machine-executable action by a subject in a given situation. See also: obligation.

Prompt A question that can be answered with either “Yes” or “No”.

Virtual Machine Software implementation of a machine that executes computer programs.

VM See *Virtual Machine*.

Web Ontology Language A family of knowledge representation languages for specifying ontologies, endorsed by the W3C.

XSL See *Extensible Stylesheet Language*.

XSL Transformations A member of the *XSL* family, intended to transform one XML document into another.

XSLT See *XSL Transformations*.

Contents

Preface ii

Summary iii

Glossary iv

List of Figures ix

I Introduction 1

1 Introduction 2

2 Preliminaries 4
- 2.1 Introduction to MECA 4
- 2.2 The Brahms Multi Agent System 5
- 2.3 KAoS Policy Services 5

II Developing the KAoS/Brahms bridge 7

3 Analysis 8
- 3.1 Problem definition 8
- 3.2 Problem analysis 8

4 Design 10
- 4.1 Brahms / Java interface 10
- 4.2 Architecture 11
  - 4.2.1 Ontology Builder 11
  - 4.2.2 Brahms Policy Library 13
  - 4.2.3 KAoS Gateway Agent 14
- 4.3 Process flow 15

5 Test plan 17
- 5.1 Unit testing 17
- 5.2 Integration testing 17
- 5.3 Regression testing 18

6 Implementation 19
6.1 Policy Library 19
6.1.1 Using the Brahms Policy Library 20
6.2 KAoS Gateway 21
6.2.1 Core components 22
6.3 Ontology Builder 25
6.4 Test results 27
6.4.1 Unit tests 27
6.4.2 Integration tests: Bridge Testing Project 27

III Simulating the MECA scenario 29

7 Modelling in Brahms 30
7.1 Simulation members 30
7.2 Policies 30
7.3 Alternative scenarios 32
7.4 Rover requesting the route 33
7.5 Rover receiving new information 33
7.6 Assisting Benny with his spacesuit failure 36

8 Human-in-the-loop 38
8.1 Analysis 38
8.2 Design 38
8.2.1 Interface 39
8.2.2 Simulation in real time 40
8.3 Implementation 42

IV Conclusions, Recommendations and Future work 44

9 Conclusions 45
9.1 Work delivered 45
9.2 Tentative results 46

10 Recommendations 47
10.1 Recommendations for Brahms 47
10.2 Recommendations for KAoS 49

11 Future work 50
11.1 KAoS/Brahms bridge 50
11.2 Ontology Builder 52
11.3 MECA Simulation 52

V Appendices 53

A Requirements analysis document 54
A.1 MECA simulation 54
A.2 KAoS/Brahms bridge 56
A.3 Graphical User Interface 57

B Scenarios **59**
B.1 First iteration scenario 59
B.2 Second iteration scenario 61

C Requirements evaluation **63**

D Installation guide **65**
D.1 Installation of the KAoS/Brahms bridge 65
D.2 Running the MECA simulation 67
D.2.1 Starting KAoS with the MECA ontologies 67
D.2.2 Running the ePartner GUI 67
D.2.3 Running in the Brahms Composer 67
D.3 Using the KAoS/Brahms bridge 68
D.3.1 Using the bridge in a Brahms model 68
D.3.2 Using the Ontology Builder 68

Bibliography **68**

List of Figures

2.1 MECA as ubiquitous ePartner 4

3.1 Schematic overview of the querying process the KAoS/Brahms bridge will be used in 9

4.1 System overview of the KAoS/Brahms bridge, in context of a Brahms application relying on KAoS policies 12
4.2 High-level UML sequence diagram of the querying process 16

6.1 Brahms source code of an agent using the Policy Library 20
6.2 Brahms source code of an agent using the Policy Library (continued) 21
6.3 UML class diagram of the KAoS/Brahms bridge 22
6.4 UML sequence diagram showing how a policy request is handled in the KAoS Gateway 24
6.5 UML sequence diagram of handling obligations 26
6.6 System overview of the Ontology Builder 27

7.1 Class diagram of all agents and agent groups in the Brahms model 31
7.2 Rover driving into crater because Brenda overrides safety policy 32
7.3 Agents collaborating when new information arrives 34
7.4 State diagram when new information arrives at the eRover . . . 35
7.5 eRovers activity stack when new information arrives 35
7.6 Brenda collaborating with her ePartner to help Benny 36
7.7 eBrenda interrupting Brenda, caused by inform from eRover . . . 37

8.1 Event processing sequence when a human actor is simulated . . . 39
8.2 Event processing sequence when the interface is attached directly to the ePartner 39
8.3 Event processing sequence when the interface is attached to a separate agent 40
8.4 Event processing sequence when the interface is attached to a separate agent, leaving simulation code unchanged 40
8.5 Touch screen user interface for Brenda 43
8.6 Prompt from rover on touch screen user interface 43

B.1 Map of the environment used in the scenarios 60

Part I

Introduction

Chapter 1

Introduction

Future manned missions to the Moon and Mars are planned, but while mankind has gained significant experience in space flight since Yuri Gagarin became the first man in space in 1961, space will always be a dangerous environment. Space missions become more difficult with increasing distance from Earth, as communication with mission control takes ever more time and consumes more energy. Up to forty minutes can go by between sending a message from Mars and receiving the answer from Earth.

To counter these difficulties, members of future space missions need to be more autonomous, while still being able to accomplish tightly planned missions. To assist with the increased autonomy the Mission Execution Crew Assistant has been designed. This is a system that "supports (groups of) humans and machines to act in a distributed, autonomous but cooperative way" [9, 10]. The European Space Agency is funding the aptly named MECA project to make such a smart assistant become reality. These assistants are also known as "ePartners", short for "electronic partners".

The current focus of the MECA project is to define a requirements baseline for an assistant as just described [8]. By analysing simulations of scenarios that may occur during space missions, this baseline is refined. There are several ways in which such a simulation experiments can be set up.

One way to do so is to use a multi-agent system, in which the actors from the scenario are modelled as agents. These agents can be defined using the *Brahms* language. This language has been designed specifically to model the contextual behaviour of groups of people, called work practice [11]. However, while Brahms is suitable for running simulations, it is not built to store and enforce the policies needed to safeguard crew members. These policies can be stored and controlled by the KAoS policy framework. Therefore, a bridge between Brahms and KAoS should be realised.

Another way of simulating scenarios is to use human-in-the-loop experiments. In such an experiment, several human subjects play the role of actors in the scenario, under supervision of an experiment leader. These experiments are well suited to evaluate human-computer and human-robot interaction, which are important factors in the design of MECA.

In this report we present a simulation of a MECA scenario, powered by Brahms and KAoS. To achieve this, a subsystem has been developed to allow agents in a Brahms multi-agent system to query KAoS policies. Because the development of the bridge and the simulation proceeded quickly, we were able to further modify the simulation for use in a human-in-the-loop experiment as well.

In the remainder of part I the concepts used to develop the bridge and the simulation and their context are described more elaborately. Part II presents the software-engineering process for the KAoS/Brahms bridge, which allows Brahms agents to query KAoS policies. With the bridge available, the simulation of a MECA scenario using KAoS and Brahms is explored in part III. This part also contains the modifications done to support the human-in-the-loop experiment. Finally, our conclusions are presented in part IV.

Chapter 2

Preliminaries

In this project, Brahms and KAoS are used to simulate a Mission Execution Crew Assistant (MECA) scenario. First, we will give an introduction into these topics. The goals and current state of the MECA project is given in section 2.1. Then, an introduction to Brahms is given in section 2.2. Finally, we describe the KAoS framework, and how it's used to enforce policies in section 2.3.

2.1 Introduction to MECA

The Mission Execution Crew Assistant project is an European Space Agency (ESA) research project "that empowers the cognitive capacities of human-machine teams during planetary exploration missions in order to cope autonomously with unexpected, complex and potentially hazardous situations" [10]. MECA uses the concept of personal electronic partners (or ePartners) which act in an ubiquitous computing environment (represented in figure 2.1) that enhances the cognitive capabilities of human-machine teams.

Figure 2.1: MECA as ubiquitous ePartner

In the MECA project human factors are analysed using simulation-based

testing of scenarios. These scenarios involve different actors and focuses on the interactions between them [10]. This thesis focuses on a scenario involving two astronauts, Benny and Brenda, who are travelling to a habitat aboard a rover. Brenda's ePartner assists in diagnosing and monitoring Benny, who suffers from hypothermia due to a space suit failure. By analysing MECA in context of scenarios like this the requirements baseline for the eventual assistant to be sent into space can be further refined.

The MECA project consists of three phases. In the first phase, an initial version of both user and software requirements have been defined. These requirements were refined in the second phase, during which a proof-of-concept demonstrator of the MECA system was built and used in experiments. As of 2008, the MECA project has entered Phase 3, which includes several follow-up projects such as participation in the MARS 500 experiment [3].

2.2 The Brahms Multi Agent System

Brahms is "a set of software tools to develop and simulate multi-agent models of human and machine behaviour" [2], developed at NASA's AMES Research Center. These tools include a compiler and a virtual machine for the Brahms Language, in which the models are defined.

One of the most distinguishing characteristics of Brahms is that it is specifically designed to allow the modelling of *work practice*. Brahms' creators define work practice as follows:

> The collective performance of contextually situated activities of a group of people who coordinate, cooperate and collaborate while performing these activities synchronously or asynchronously, making use of knowledge previously gained through experiences in performing similar activities [11].

One of the key phrases in this definition is *activities*. Activities form the smallest unit of execution in a Brahms model, rather than plans such as in common BDI systems. To make sure models remain workable however, the right granularity must be obtained when activities are specified. The most suitable level of detail depends on the nature of the model and lies between the high level formal process models on one side and the low level cognitive models on the other.

2.3 KAoS Policy Services

KAoS is a set of platform-independent services developed by the Institute for Human and Machine Cognition (IHMC). The KAoS Policy Service allows the definition of policies, which can be enforced in both agent and traditional distributed systems [13]. Policies are defined as an enforceable, well-specified constraint on the performance of a machine-executable action by an actor in a given situation [1].

The KAoS Policy Service distinguishes between authorisations and obligations, as defined by Sloman et al. [4]. An authorisation is a constraint that either permits (*positive authorisation*) or forbids (*negative authorisation*) some action. An obligation is a constraint that requires the agent to perform some action when a state- or event-based trigger occurs.

When time-sharing a car, for example, an obligation could be that every user has to re-fill the tank after using it. This can be described in a KAoS obligation as follows:

```
CarUser is obligated to start performing FillTheTank
  which has any attributes
after CarUser finishes performing Drive
  which has any attributes.
```

Restrictions can also be added for example by setting the `fillCompletely` attribute property of the `FillTheTank` action. The following obligation states that the agent needs to fill the tank completely.

```
CarUser is obligated to start performing FillTheTank
  which has attributes:
    the FillTheTank.fillCompletely value is true
after...
```

The CarUser agent can query KAoS by creating an Action Instance Description (AID). For this example the AID contains that the CarUser has finished performing Drive. In response to this AID, KAoS will return a new AID containing that the CarUser needs to perform FillTheTank. The CarUser agent will then have the choice of either performing this action or ignoring the obligation.

KAoS does not force agents to obey policies, it only determines if they apply and what their implications are. This ensures that agents remain autonomous, which is after all one their defining properties according to Wooldridge [17].

While KAoS can work in a multi-agent environment it is not itself a complete multi-agent programming language: for example, it only determines whether an action may or may not be executed, but does not support their actual execution. Performing actions always needs to be done by the system querying KAoS, be it a Java program or a multi-agent framework such as Brahms.

The KAoS Policy Service is relevant to the MECA project, because it presents an intuitive way to define safety rules. Without KAoS these rules would either have to be hard-coded in the agents or a completely new policy system would have to be developed. Earlier research at TNO has demonstrated that KAoS is capable of representing the rules that are needed for MECA [14].

Part II

Developing the KAoS/Brahms bridge

Chapter 3

Analysis

To allow Brahms and KAoS to be used in a single simulation, both frameworks must be capable of communicating with each other. However, because of the different communication languages and protocols in Brahms and KAoS some method of translation is needed.

To specify exactly what this translation entails, requirements have been formulated. The results of the requirements elicitation can be found in appendix A. Based on these requirements, this chapter first gives a definition of the problem that needs to be solved in section 3.1. This definition is then further analysed in section 3.2 to identify the subsystems necessary to solve this problem.

3.1 Problem definition

To allow Brahms and KAoS to be used in a single simulation, both frameworks must be capable of communicating with each other. In other words, the following is needed:

> *A system that allows agents in a Brahms multi-agent system to query* KAoS *policies.*

3.2 Problem analysis

Based on the definition given above, the purpose of the system is to support *querying* from Brahms to KAoS. The process of querying between two different systems generally includes the following steps:

1. Receive a request specified in the host language.
2. Translate the request to the language used by the target system.
3. Send the translated request to the target system.
4. Receive the response from the target system.
5. Translate the response back into the language of the host system.
6. Send the request to the host system.

Applying this to the context of communication between Brahms and KAoS yields the following steps:

1. Receive a request from Brahms
2. Translate the request to KAoS' query language.
3. Send the translated request to KAoS.
4. Receive the response from KAoS.
5. Translate the response back into Brahms concepts.
6. Send the concepts back to Brahms.

Because KAoS and Brahms use different languages, some sort of bridge will be needed to allow the two to communicate. A schematic overview of the process, including this bridge, is given in figure 3.1.

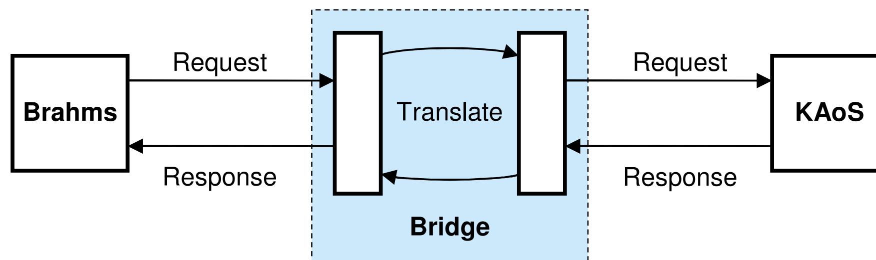

Figure 3.1: Schematic overview of the querying process the KAoS/Brahms bridge will be used in

This process serves as a template on which the system can be modelled. On closer inspection of the process a number of system requirements can be derived. For policies to be checked they first need to be defined. Because KAoS requires an ontology of the model to define policies for it, this is a non-trivial task. Hence the system first needs to ensure that the KAoS ontology is a correct representation of the used Brahms model.

Once the polices are defined, two more capabilities are required to allow agents to check them while the system is running. This means the system needs to enable Brahms agents to formulate queries and to handle the answers they receive. Finally, these queries need to be translated from Brahms into a format KAoS can handle.

Now that a general outline of the necessary capabilities has been made, the exact way they operate must be defined. This will be done in the design, presented in the next chapter.

Chapter 4

Design

This chapter describes the design of the KAoS/Brahms bridge, based on the analysis in chapter 3.

The common link between Brahms and KAoS is that they both provide interfaces to communicate with programs written in Java. KAoS offers one interface for the communication between Java programs, however Brahms offers two. Because the two Java communication interfaces in Brahms pose great differences for the design of the bridge, this decision will be addressed first in section 4.1.

Following this decision, the architecture of the system is described in section 4.2. Finally in section 4.3 the design is placed in context of the process described in the analysis.

4.1 Brahms / Java interface

Brahms provides a Java interface to communicate with the world outside the simulation [15]. There are two ways in which this interface can be used: either by calling a Java activity within a Brahms agent, or by creating an external agent entirely implemented in Java. This section compares these two alternatives and then determines which of the two is the best solution for the KAoS/Brahms bridge.

To use Java activities in Brahms, the programmer creates a Java class with a special method. This method is invoked by the VM when the Brahms agent invokes the activity. Java activities can be used anywhere ordinary activities can be used.

When using an external Java agent on the other hand, a Java class is created that implements a specific interface. This class is then registered as an external agent in the Brahms model. When the Brahms VM runs the model, it instantiates the class and informs it of any events happening in the simulation. Other agents in the environment can communicate with the external agent using the standard communication activities in the Brahms language.

The main advantages of using activities are that they do not require a complex communication protocol and that the activity semantics are very similar to object-oriented method calls, which makes them intuitive to use. However, the system needs to work in an agent-oriented environment, which poses differ-

ent requirements. Apart from that, activities cannot use state to speed up the process or help in debugging because new instances are created for every call. Finally, using activities results in a great number of access points to KAoS: when an agent invokes an activity, that activity communicates with KAoS, which means there are at least as many access points as there are agents.

This does not occur when a external agent is used, because in that case all communication is run through one agent especially designed for being the bridge between Brahms and KAoS. This has a number of advantages. Because the bridge is an agent itself, it is a cleaner conceptual match with the surrounding system. Development is also easier because an agent can easily be replaced by a stub while working on other parts of the system. Furthermore an agent has the advantage that it has persistent state while the system runs, allowing it to use resources more efficiently. However, this does come at the price of a more complex system, because communication protocols need to be designed.

Conclusion
Based on the previous discussion, we decide to use an external agent. The most important reasons are the ease of developing and debugging, and the cleaner conceptual role of the bridge agent.

While complexity is an issue that needs to be addressed, the task at hand is relatively simple. Essentially, the bridge only needs to translate information and pass it on. Each action of the bridge itself is almost atomic, there is very little coupling between consequent requests. This lack of interdependencies makes it significantly easier to deal with the multi-threaded environment.

The communication protocol needs to be implemented, but the use of Java activities would have required some sort of protocol as well. We consider the slightly higher price paid for an inter-agent protocol compared to an activity protocol worth the cleaner conceptual view of the system as a whole.

4.2 Architecture

The system consists of three main components: a Brahms Policy Library, the KAoS Gateway Agent and the Ontology Builder. The relations between these components, as well as the relations with KAoS and the Brahms model are shown in figure 4.1.

4.2.1 Ontology Builder

When the KAoS Policy Services are used to enforce policies on a Brahms model, a mapping is needed between concepts in Brahms and the corresponding concepts used by KAoS. Both this mapping and the ontology used by KAoS can be manually constructed by the Brahms modeller, but it is a complex task to keep the model and ontology consistent, especially within a team project.

Policies in KAoS are defined over an ontology, which is a "formal representation of a set of concepts within a domain and the relationships between those concepts" [16]. As discussed in section 2.3, KAoS uses the Web Ontology Language (OWL) to represent these ontologies. The OWL standard as specified in [5] does not enforce the way in which a ontology is serialised, however using a RDF/XML syntax seems to be common practice. Herein lies the power and

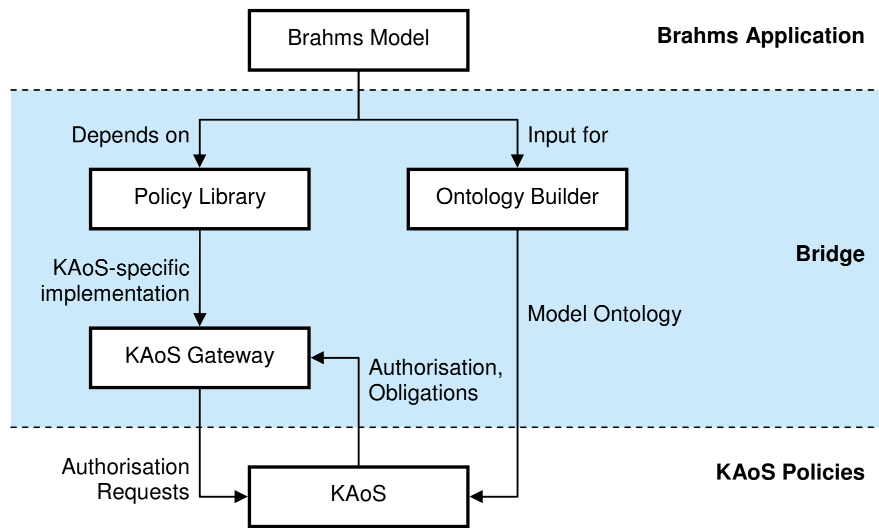

Figure 4.1: System overview of the KAoS/Brahms bridge, in context of a Brahms application relying on KAoS policies

simplicity of our Ontology Builder, because a compiled Brahms model is also represented in a well specified XML format.

There are two different ways to construct the mapping between Brahms and KAoS. An ontology can be constructed from a set of Brahms models, or Brahms models can be generated from an ontology. We have chosen the option to construct an ontology from a set of Brahms models. Brahms models contain more than just a representation of policies; they contain code that specifies when and how workframes/activities are performed. Generating Brahms models from an ontology would mean that code gets lost every time a new ontology is created. The loss when having to redefine policies is a lot smaller since policies in KAoS can easily be saved and restored if they are still applicable. Secondly designing the system this way allows for previous Brahms projects to be extended with policies more easily. A modeller would generate an ontology from the Brahms models and can add code to deal with policies incrementally to prevent his code from breaking. Third, modelling your agents in Brahms is a lot more explicit than modelling a formal set of concepts and their relationships allowing the modeller to think more about the work practice in the scenario that needs to be modelled than the actual formal representation.

For the three previously mentioned reasons we have chosen to implement a tool that generates both the ontology and the mapping from the Brahms model. This enables the modeller to automatically update the ontology whenever the model changes.

To achieve this, the Builder needs to load the model and extract an ontology defined in the Web Ontology Language (OWL). After this OWL file has been loaded into KAoS, policies can be defined.

4.2.2 Brahms Policy Library

Even when policies can be defined, Brahms agents need to be able to query them before they can be of any use. This section first discusses how the second capability described in the analysis can be fulfilled: to enable Brahms agents to formulate queries and handle the response to those queries. Once a request is formulated, it must be sent to the actual bridge. There are two ways to do this, which will be discussed in the second part of this section.

Formulating Brahms requests

To allow Brahms to query KAoS for policy decisions about the execution of activities, it will first have to be able to formulate these queries. This can either be done by providing an API to handle the communication with the bridge, or by letting the application programmer arrange communication with the bridge himself.

When using an API, all an application programmer has to do is define the actions for which authorisation is needed and their properties. The API should then take these action descriptions and send them to bridge for further processing. The advantage of this approach is that it takes a lot of work off the application programmers' hands. They do not need to worry about the low-level workings of the communication between Brahms code and the bridge. It also makes it easier to re-use code and because all function calls are grouped within the API, functionality is easier to document.

A potential downside to this method is that creating a separate API takes more time. However, it is expected that there will be quite a number of policy requests. If specific code were to be written for each request, it would quickly become bothersome to maintain it in the face of changes. This maintenance time is expected to exceed the time needed to create the API with only a few calls.

If the application programmer arranges all communication by himself, he has greater control over the processing of responses to his requests. As just mentioned however, this comes at the price of more time spent on maintenance. Also, even if an API is provided, a programmer can still bypass it and just communicate with the bridge directly.

Conclusion
Taking the preceding into account, we consider it justified to create an API in Brahms for use by application programmers, the Brahms Policy Library. The main argument for this decision is that abstraction makes the development of Brahms models that rely on KAoS a lot easier, which is one of the main goals of this project. Also, the initial investment of creating an abstract library is expected to pay itself back after only a few uses.

Communicating action attributes

An important aspect of KAoS policies is the ability to authorise actions based on their properties. To unlock the full potential of policies, these properties must therefore be communicated to KAoS when a Brahms agent issues a request to perform an action. This can be done in two ways, which will be discussed in this section.

One way is for the Brahms agent to send the information encapsulated in a message as defined by the FIPA Agent Communication Language [6]. In this case, a new Brahms class is created for each kind of action, containing the attributes that are relevant to the authorisation of the action. When an agent wants to obtain authorisation to perform an action, it constructs a new instance of this class, sets the values for the relevant attributes and sends it to the KAoS Gateway Agent, encapsulated in a FIPA message. The KAoS Gateway Agent passes the action and its attributes on to KAoS. The result of the policy check is written back to the object, which the agent can then use to determine what it should do.

A second option is to merely inform the KAoS Gateway Agent what kind of action needs to be done, and have it find out the properties to that action by extracting information from the requesting agent. Here, the Brahms agent merely sends the KAoS Gateway Agent a message in plain text indicating what action it wants to perform. The KAoS Gateway Agent then extracts the required properties from the agent using the VM access controls in the Brahms Java API and sends the fully specified request to KAoS. The authorisation result is written back to an attribute in the agent, which it can consequently use to determine what is should do.

The main advantages of the FIPA-approach are that it allows better encapsulation of information and uses a standard protocol. All data relevant for a policy request is contained within the object sent in the FIPA message. This also makes it easier to manage multiple pending request, as there are no shared fields between them. Using FIPA ACL makes it easier for other agent systems in Brahms to communicate with the library.

When the data are extracted from agents the communication protocol is a lot simpler. However, this comes at a price. The most important disadvantage is that extracting beliefs from agents violates the semantics of agent-oriented programming: extracting information without consent violates an agent's autonomy. It also severely limits the amount of control an agent can exercise over the requests it issues. Finally interfacing with new systems becomes a lot more difficult, because they have to structure there own beliefs to match the expectations of the library.

Conclusion

With the preceding discussion in mind, we decide to use the FIPA-approach. The main reason for this is the cleaner conceptual structure of the resulting system, which we expect to make development easier when the system becomes more complex and will also facilitate integration in future projects.

4.2.3 KAoS Gateway Agent

Once requests have been formulated in the Policy Library, they need to be sent to KAoS. This is done using the KAoS Gateway Agent, which serves as a translation gateway to KAoS, in accordance with the final capability identified in the analysis. Translation is necessary because the Brahms agents produce Brahms objects, while KAoS requires so-called Action Instance Descriptions. Also, KAoS uses Action Instance Descriptions to respond to queries, and these will have to be translated back into Brahms objects to be useful to the original requesting agents.

If the full potential of KAoS policies is to be realised, this translation needs to be as complete as possible. This means that attributes of actions need to be taken into account when translating from Brahms to KAoS. However obligations and restrictions on attributes of obligated actions must also be expressible in Brahms objects when a response is to be sent back to an agent.

This indicates the need for two subsystems in the KAoS Gateway agent: one to deal translation, and one that will handle the communication of the translated objects, both to KAoS and to Brahms.

4.3 Process flow

Now that the main components of the system are specified, they can be inserted in the process described in the analysis in chapter 3. A high-level UML sequence diagram of the querying process using the components just discussed is given in figure 4.2.

1. A Brahms agent asks the Policy Library to formulate a policy request.
2. When the request is ready the agent asks the library to send it.
3. The library sends the request to the KAoS Gateway Agent.
4. The KAoS Gateway Agent translates the Brahms message into a KAoS Action Instance Description.
5. Once this is done, the AID is sent to KAoS, which then returns the policy decision.
6. This decision is translated back into Brahms objects in the KAoS Gateway Agent, and returned to the Brahms agent who initiated the query.

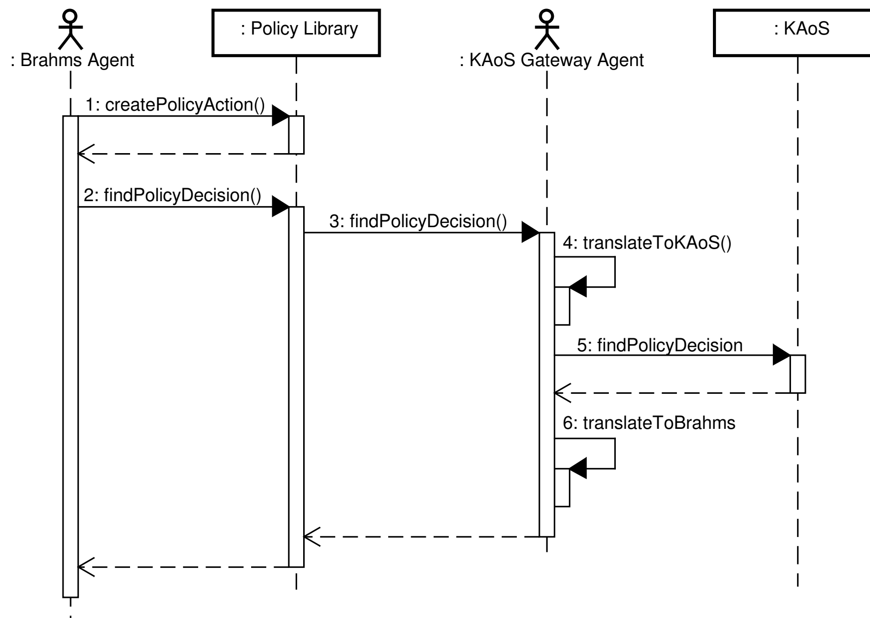

Figure 4.2: High-level UML sequence diagram of the querying process. To query KAoS for a policy decision, a Brahms Agent first uses the Policy Library to formulate the query (1). This query is then dispatched to the Policy Library (2), who sends it to the KAoS Gateway Agent (3). The Gateway first translates the Brahms request into a KAoS Action Instance Description (AID, 4) and then sends this AID to KAoS (5). The response from KAoS is translated back into Brahms (6) and is then sent back to agent who sent the query.

Chapter 5

Test plan

The KAoS/Brahms bridge is tested using various methods, which will be discussed in this chapter. Sections 5.1, 5.2 and 5.3 will respectively deal with unit, integration and regression testing.

5.1 Unit testing

Unit testing is used to test the functionality of the individual low level components which the system consists of. By testing at a low level faults can be detected quickly, which helps speed up the development process. Testing these components is done using JUnit, a Java library developed specifically for this purpose.

The following components will be subjected to unit tests:

1. The KAoS AID Builder
2. The Communicative Act Handler
3. The Policy Action model used in the bridge

Test cases for these components are formulated based on the functional requirements specified in appendix A. Another source for test cases are the system models developed in chapter 6. Testing will be deemed sufficient if it achieves 100% statement coverage of the classes listed in the previous paragraph. The coverage will be determined using the Eclemma coverage plug-in for Eclipse.

An exception of the 100% coverage criterion is made for the statements used for handling Brahms Exceptions. These exceptions are omitted from the coverage criterion because in most cases they are thrown only when the underlying Brahms VM is starting up or shutting down. Here, the exception has nothing to do with the implementation of the bridge. In cases where an exception may be caused because of the implementation, test cases will be added to check the behaviour.

5.2 Integration testing

Even when the separate units of the system function perfectly, faults may still occur when components are put together, for example due to errors in commu-

nication. These faults can be detected with *integration tests*, which make sure that groups of components within the system function as they should.

Because the KAoS/Brahms bridge is composed out of many separate systems, it is hard to use automated testing to verify the correctness of the interfaces between them. Testing the interface between the KAoS Gateway Agent and KAoS itself can be done entirely in JUnit, but requires the KAoS back-end to be available during tests. This would also require the creation and automatic loading of various policies created for testing alone.

Testing the interface between the Brahms Policy Library and the KAoS Gateway Agent is even more problematic. In this case the code which invokes the interface is not even written in Java but in Brahms, which makes it difficult to use JUnit for automated testing. While testing frameworks for multi-agent systems are being developed (see [12] for example), they focus on pure BDI systems, while Brahms is an activity-oriented system.

To be able to test the integration of the KAoS Brahms bridge in spite of these issues, an alternative approach can be used. The bridge can be integrated in a small test project, in which the various API methods offered by the Brahms Policy Library are all used. The policy library will forward these requests to the gateway agent, which in turn will translate them to KAoS. Using this approach, the entire bridge is tested for faults in interfaces. While the results of these tests cannot be verified automatically, manual inspection will show whether the bridge is functioning correctly.

5.3 Regression testing

Adding new functionality to a piece of software usually involves changing code that was written earlier. If the developer is not careful with these changes, he might accidentally introduce new faults in functionality that was working fine previously, or re-introduce bugs that were fixed earlier. To detect the re-occurrence of bugs that were fixed earlier, *regression testing* is used.

The Brahms Virtual Machine (VM) logs of earlier iterations of the model form a source of information that may be used for regression tests. These logs show a certain ordering in which actions must take place. Automatic testing can be used to determine whether these orderings are still adhered to when the model has changed.

To test this, VM logs from earlier runs can be used to determine patterns that should always occur in a certain order. These patterns are then captured in a test-case that can automatically determine whether a given VM log contains the required pattern or not.

Following a change, the model is rerun to generate a new VM log. The test cases are then executed to verify that the patterns that should occur are still present in the correct order. If this is not the case, some important behaviour has apparently changed unintentionally, indicating a regression bug.

Chapter 6

Implementation

The KAoS/Brahms bridge consists of three main components: the Policy Library, the KAoS Gateway and the Ontology Builder. The implementation of these components is discussed in sections 6.1, 6.2 and 6.3 respectively.

6.1 Policy Library

The Policy Library is written entirely in Brahms and provides an API which application programmers can depend on to interface with policy services. The library handles policy requests by delegating them to an agent which can query the policy service directly. Communication between the library and the agent is done using FIPA communicative act messages. This means that the library can interface with any kind of policy service, provided there is an agent capable of handling these messages.

The API itself introduces three concepts: the `PolicyActor` agent group and the `PolicyAction` and `Restriction` classes. To allow an agent to communicate with the policy provider, it needs to become member of the `PolicyActor` group. This group provides methods to request authorisation for specific actions and makes sure the agent is made aware of any obligations the execution of these actions may entail.

Actions are represented as objects inheriting from the `PolicyAction` class. The minimum content of a `PolicyAction` instance is the name of the agent who wants to execute the action. However, an application modeller can create new subclasses which specify any number of additional attributes, all of which can be used when defining policies. Also a `PolicyAction` instance will always include an attribute reflecting its authorisation state. Application code can monitor the value of this attribute to determine the result of an authorisation request.

`PolicyAction` instances are used for both communication to and from the policy service provider. As was just described, agents in the application may create new instances to request authorisation. However, when this action requires obligations to be performed, these are represented as `PolicyAction` instances as well. The Policy Library ensures that these obligation instances are coupled with the original `PolicyAction` request using a `hasObligation` relation.

The library also makes sure any restrictions on attribute values in the obligated actions are communicated to the agent. This is done by attaching

Restriction objects to the obligation action with a hasRestriction relation. Because restrictions are related to the attributes of the specific subclasses of PolicyAction defined in the application, the modeller of the application agent will have to write custom code to make sure the agent obeys these restrictions.

6.1.1 Using the Brahms Policy Library

We will discuss the Brahms source code for a simple Brahms agent that uses the Brahms Policy Library. Listing 6.1 and 6.2 contains the source code of this agent. At line 1, we define a Robot that is member of the agent group PolicyActor. This membership gives the agent activities to obtain authorisation and obligations from the policy service. At line 3 we set an initial belief of the agent. The agent now believes that KaosGatewayAgent is the policy service.

The Robot has only one workframe, starting from line 6: it wants to move to HomeLocation. To query the policy service for authorisation for this move, the agent has to create an instance of a PolicyAction (consult section 6.1 for more details). Since the agent wants to move, it creates an instance of a subclass of PolicyAction called MoveAction. This is done using the library call createPolicyAction at line 10. The created object instance is bound to a variable called moveToHomeAction. To specify to which location the agent wants the move, the agent has to create a belief about this location. The conclude statement at line 11 creates this belief. Now the PolicyAction is defined it is ready to be send to the policy service. In this example, we create a composite activity tryToAuthoriseMoveAction (see listing 6.2) that queries the policy service and handles the response.

```
1  agent Robot memberof PolicyActor {
2    initial_beliefs:
3      (current hasPolicyGateway KaosGatewayAgent);
4
5    workframes:
6      workframe wf_moveToHome {
7        variables:
8          forone(MoveAction) moveToHomeAction;
9        do {
10         createPolicyAction(MoveAction, moveToHomeAction);
11         conclude((moveToHomeAction.targetLocation
12                        = HomeLocation));
13         tryToAuthoriseMoveAction(moveToHomeAction);
14       }
15     }
```

Figure 6.1: Brahms source code of an agent using the Policy Library

Once the composite activity tryToAuthoriseMoveAction is called, the workframe wf_askForAuthorisation executes. At line 23, this workframe uses the library activity findPolicyDecision. This activity sends the MoveAction to the policy service and waits until the authorisation result is known. When positive authorisation is received, the workframe wf_handlePositiveAuthorisation executes because its precondition evaluates to true. This workframe performs all the actual work: the obligations are executed and a doMove activity is called.

A description of how to execute obligations is outside the scope of this example, but it is provided in the demo project described in section 6.4.2.

In case of negative authorisation, workframe `wf_handleNegativeAuthorisation` executes. An actual agent could try obtain authorisation for some other action at this point, while this example agent executes a `callForHelp` activity. Note that the agent could disobey the policy by executing the move anyway in this workframe.

```
16   activities:
17     composite_activity
18         tryToAuthoriseMoveAction(MoveAction moveAction) {
19       workframes:
20         workframe wf_askForAuthorisation {
21           repeat: false;
22           do {
23             findPolicyDecision(moveAction);
24           }
25         }
26
27         workframe wf_handlePositiveAuthorisation {
28           when(knownval(moveAction.isAuthorized = true))
29           do {
30             executeObligations(moveAction);
31             doMove(moveAction);
32           }
33         }
34
35         workframe wf_handleNegativeAuthorisation {
36           when(knownval(moveAction.isAuthorized = false))
37           do {
38             callForHelp();
39           }
40         }
41     }
42 }
```

Figure 6.2: Brahms source code of an agent using the Policy Library (continued)

Full source code including documentation is provided in the KAoS/Brahms bridge distribution.

6.2 KAoS Gateway

The KAoS Gateway is the KAoS-specific implementation of the Policy Library used to communicate with KAoS. This subsystem consists out of five core components: the KAoS Gateway Agent, the Communicative Act Handler, the KAoS AID Builder, Obligation Handler and the KAoS Policy Gateway.

The class diagram in figure 6.3 shows the relations between these five components. This diagram also shows two gateway interfaces, `IBrahmsGateway` and `IPolicyGateway`, which are extracted from respectively the KAoS Gateway Agent and the KAoS Policy Gateway. The main purpose of this extra layer

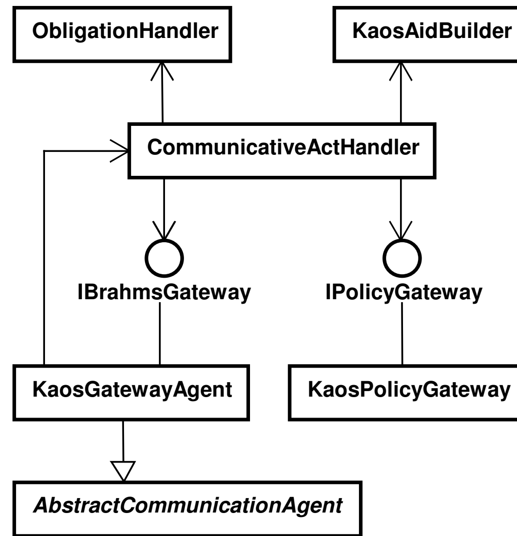

Figure 6.3: UML class diagram of the KAoS/Brahms bridge

of abstraction is to facilitate testing. Because of these interfaces the full-blown Brahms and KAoS implementations can be replaced by simple stubs. This makes it easier to verify whether the implementation of the agent itself works as it should, without the need of running the actual simulation.

6.2.1 Core components

KAoS Gateway Agent

The KAoS Gateway agent serves as an entry point for incoming requests and as a gateway to send messages back to other Brahms agents. To fulfil its role as entry point, it extends the `AbstractCommunicationAgent` class from the Brahms Java API, which gives the agent the ability to communicate using communicative acts. When the gateway agent receives a communicative act, it is immediately handed off to the Communicative Act Handler. Outgoing responses are sent using the Abstract Commmunication Agent functionality as well, but to facilitate testing this has been extracted into an additional interface, `IBrahmsGateway`.

Communicative Act Handler

The Communicative Act Handler directs the process of handling an incoming message from the moment of receipt to the sending of the last response. Figure 6.4 shows how the Communicative Act Handler cooperates with the KAoS AID Builder and the KAoS Gateway Agent to handle a simple authorisation request.

First, the Java representation of the Policy Action the requesting agent wants authorisation for is extracted from the incoming message. This action is then translated into a KAoS `ActionInstanceDescription` (AID) using the KAoS Action Instance Description Builder described in the next paragraph. After it is constructed, this AID is handed off to KAoS, which returns an authorisation

result. Finally, this authorisation response is translated back into the Brahms and returned to the agent who requested the authorisation. If the response contained obligations, these are translated into Brahms objects by the Obligation Handler and included in the response to the requester.

KAoS AID Builder

The KAoS Action Instance Description Builder is used to convert Brahms Policy Action objects in to the Action Instance Descriptions used by KAoS to determine its policy decisions. An AID contains three important pieces of information: the name of the instance (agent), the name of the action he wants to perform and any properties further specifying this action.

The name of the agent performing the action is taken from the `onBehalfOf` attribute which any Policy Action must specify. The action name is extracted from the type name of the incoming Policy Action object. Translating the properties takes a little more work.

First, a list of all attributes of the received Policy Action is retrieved using the Brahms JAPI. For each attribute in this list a new property is added to the AID. This property maps the fully qualified name of the attribute (including package and class name) to the string representation of its value. This must be a string representation, because KAoS properties can only have string values. For example, if an instance of `com.example.TestAction` has an attribute `test` with value 42, the resulting property mapping is `"com.example.TestAction.test" = "42"`.

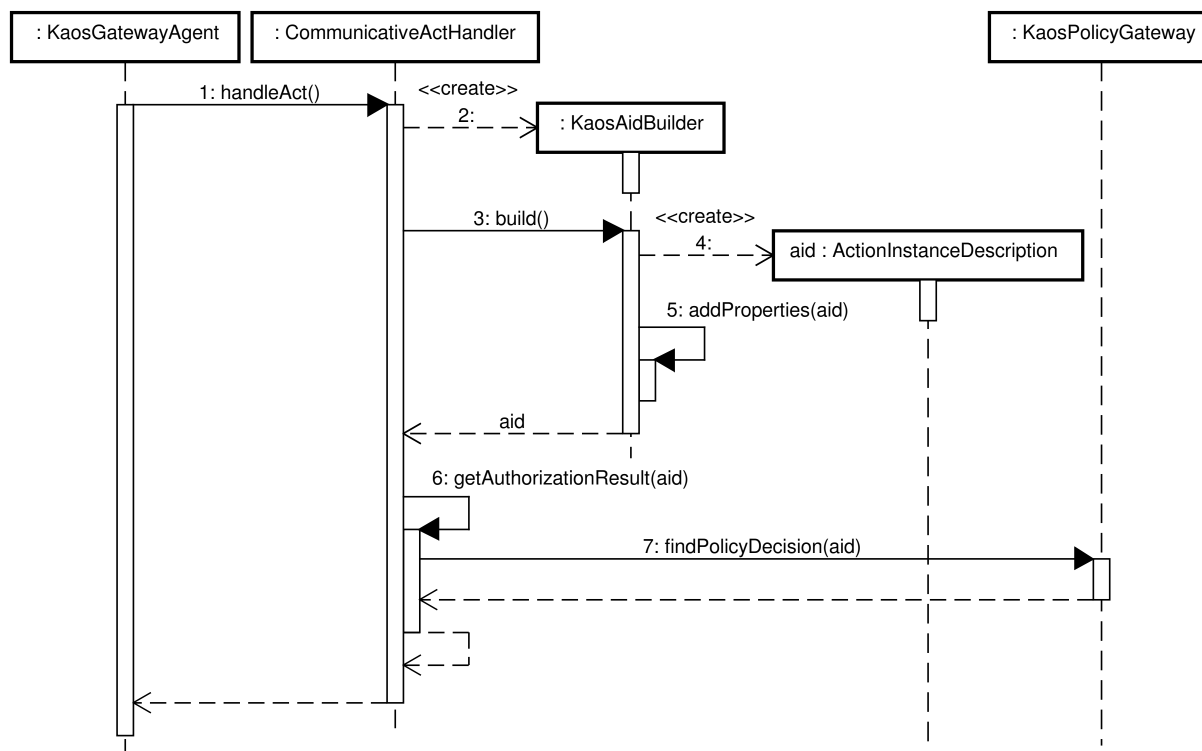

Figure 6.4: UML sequence diagram showing how a policy request is handled in the KAoS Gateway. The request is handed to the Communicative Act Handler (1), which creates an AID Builder (2) to translate the request and its properties to a KAoS Action Instance Description (4,5). This AID is then send to KAoS (6,7) and the response is returned to Brahms.

Obligation Handler

The final and perhaps most complex core component of the KAoS Gateway is the Obligation Handler. This handler is responsible for the translation of obligations that may be attached to an authorisation to perform an action. KAoS returns obligations in the form of Action Instance Descriptions, which represent the actions the agent needs to perform before or after the action it requested authorisation for. These AIDs are translated to Brahms Policy Actions using the reverse of the process in the KAoS AID Builder, shown in figure 6.5.

For each obligation the type of the corresponding Brahms Policy Action is determined by inspecting the name of the action in the AID. A Brahms object of the type specified is then created. When this is done, any restrictions placed upon values of the attributes of the Policy Action must be translated as well. For every restriction, a new Brahms `Restriction` object is created which contains the information regarding the restriction. KAoS uses an XML format that defines simple key/value pairs to specify restrictions. XML is somewhat difficult to handle in Brahms, so a helper class, the Restriction Translator, translates the XML contents and writes them to two attributes of a Restriction object.

The `attributeName` attribute determines for which field of the obligated action the restriction is valid. The second attribute, `restrictionValues`, is a map containing the translations of the key/value pairs from the XML definition. Each Brahms Restriction object is attached to its Obligation object using the `hasRestriction` relation between a Policy Action and a Restriction.

When all obligations and restrictions have been translated, they are sent back to the agent who requested authorisation. Every object is sent in a separate communicative act, because this keeps the acts themselves small and easy to handle. Sending is done using the `IBrahmsGateway` interface, which is implemented by the KAoS Gateway agent.

KAoS Policy Gateway

The KAoS Policy Gateway serves as façade for the Common Services Interface offered (CSI) by KAoS. It handles requests for policy decisions by delegating them to a `PolicyChecking` instance defined by the KAoS CSI.

6.3 Ontology Builder

Since both Brahms Compiled Code and OWL ontologies are saved in XML, a transformation between the two formats could solve the problem of manually keeping an ontology up to date. Transforming one XML format into another is done by so called XSL Transformation (XSLT), which itself is an XML-based language. By defining a XSLT stylesheet and using a XSLT parser a transformation between two XML formats is possible.

Our ontology builder consists of several XSLT files that take Brahms compiled code as input and produce several ontology files as output (Figure 6.6). A KAoS configuration file is also produced, which loads the generated ontologies with the correct settings. A modeller will only have to load the settings file into the graphical user interface of KAoS to start defining policies.

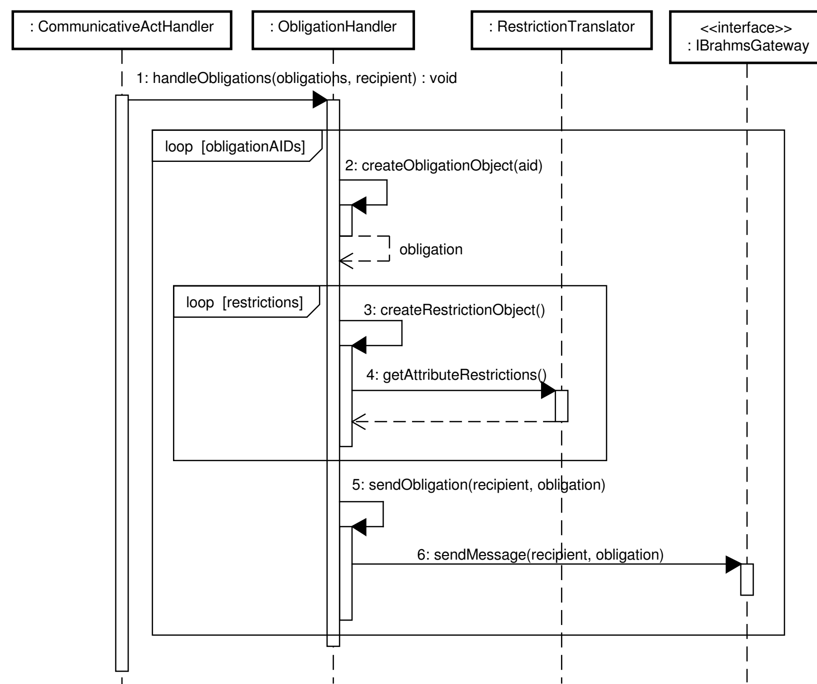

Figure 6.5: UML sequence diagram of handling obligations. This is delegated to the ObligationHandler (1), which creates new Brahms objects for every obligation (2). Restrictions are translated to objects as well (3), with the help of the RestrictionTranslator (4). The result is sent back to Brahms (5, 6).

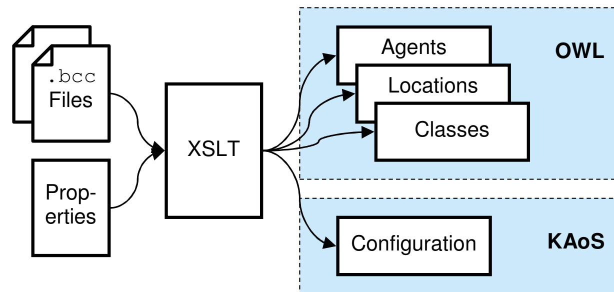

Figure 6.6: System overview of the Ontology Builder

6.4 Test results

In compliance with the test plan presented in chapter 5, tests have been made to verify that the code is working correctly.

6.4.1 Unit tests

The first set of tests is aimed at the Java implementation of the KAoS Gateway Agent. Each of the core components identified in section 5.1 is subjected to a number of JUnit test cases. These test cases are formulated to achieve 100% statement coverage, excluding code for handling Brahms Exceptions. The test coverage percentages for the core components as determined by Eclemma are listed in table 6.1.

Class	Test cases	Coverage
KaosAidBuilder	4	90%
PolicyAction	14	88%
CommunicativeActHandler	14	93%

Table 6.1: Unit test results for the KAoS Gateway Agent

The low number of test cases for the KAoS AID Builder can be explained by the fact that translating Brahms objects into KAoS AIDs turned out to be easier than expected. On inspection of the coverage report, the coverage criterion stated above turns out to be met. The code that was not covered was either dealing with Brahms Exceptions or with assertions, which Eclemma cannot interpret correctly.

6.4.2 Integration tests: Bridge Testing Project

To test whether all parts of the bridge work together correctly, a test project has been created as suggested in section 5.2. The scenario of this test corresponds to a scenario used in the KAoS documentation [7]. A robot wants to execute a move and requests authorisation for this action. The policy grants authorisation, but the robot is obligated to beep before it moves.

This scenario showcases all features of the KAoS/Brahms bridge in a simple project. Several runs were made using this project to determine if the bridge behaved correctly in variations of this scenario. This did not lead to the discovery of new bugs, showing that the bridge worked well under circumstances it was expected to be deployed in.

An additional benefit of this simple test project is that it can be used to introduce new modellers to the KAoS/Brahms bridge by showing them what the bridge can do. Some extra time has been invested to convert the test project into a full demonstration project. All relevant parts of the Brahms model code are explained with comments and documentation is provided with the demonstration on how to execute it.

Part III

Simulating the MECA scenario

Chapter 7

Modelling in Brahms

This chapter describes the Brahms implementation of the scenarios specified in appendix B, in accordance with the requirements specified in appendix A.1.

First, a general overview of the members of the scenario is given in section 7.1. The role of policies is discussed in section 7.2. With these policies in mind, the possibilities they offer for alternative scenarios are described in 7.3. Sections 7.4 and 7.5 deal with the way the rover finds its way around Mars. Finally, in section 7.6, the interaction between agents Benny and Brenda is explained.

7.1 Simulation members

The scenario to be simulated involves a large number of agents. This section describes the hierarchy in which these agents are placed and what their behaviours are. A class diagram showing the inheritance hierarchy is given in figure 7.1.

Most agents in the simulation are member of either the `Actor` or `ePartner` agent group, where each actor has exactly one ePartner and each ePartner has one actor. The ePartner serves as the actors smart digital assistant. Both `Actor` and `ePartner` can be further separated into human and rover classes. Most notable here is the introduction of two different kinds of `ePartners`, namely the `eHuman` and the `eRover`.

This difference has been created because of the different capabilities of humans and rovers. For instance a rover is not capable of putting a human in the "lateral recovery position", while on the other hand a rover can carry passengers. In the future more specialisation might be required since different astronauts can each have a different set of abilities.

7.2 Policies

The simulation makes use of two policies which are defined in KAoS. These policies are queried using the KAoS/Brahms bridge described in part II. Querying policies is one of the roles of the ePartner, therefore the ePartner agent group is member of the `PolicyActor` group provided with the Brahms Policy Library (see figure 7.1). The ontology for KAoS is created from the Brahms model using the Ontology Builder. Consult section 4.2 for an overview of the KAoS/Brahms

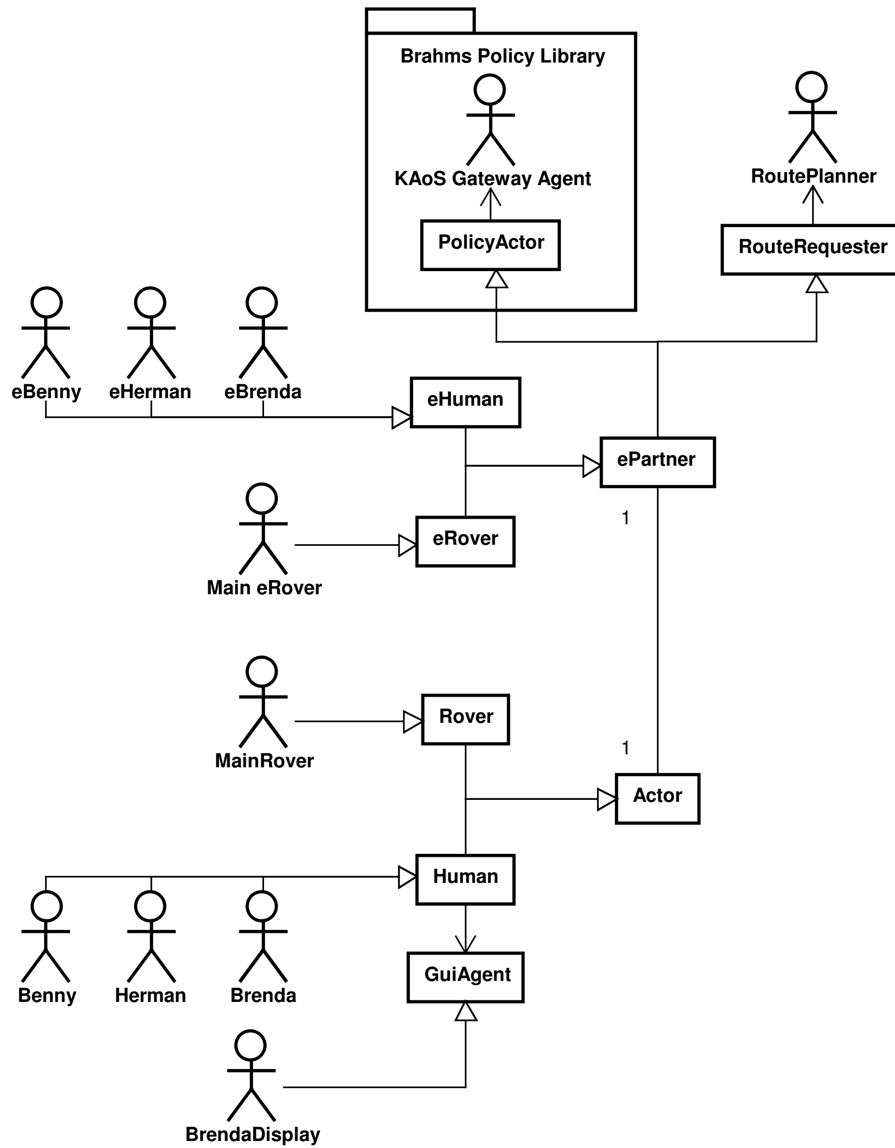

Figure 7.1: Class diagram of all agents and agent groups in the Brahms model. Agent instances (represented as actor icons) are member of agent groups (represented as class boxes). These *member of* relationships are represented as inheritance relations. Note that Brahms supports multiple inheritance.

bridge. By using the bridge to query the policies, the simulation serves as an extra test case to make sure the bridge works correctly.

The first policy in the simulation defines an obligation: the rover should be halted whenever potentially hazardous information arrives. This makes sure the rover will not continue to move when it receives information that the safety of its current route may be compromised.

The second policy is a simple authorisation: a rover may not drive through a crater which is deeper than a pre-defined value. This policy prevents a rover from falling over when it tries to descend or ascend the edge of a crater.

7.3 Alternative scenarios

To make the simulation more realistic, the occurring events should depend on whether policies are obeyed or not. More specifically, the rover should crash when it drives into a crater, as stated in requirement G.A.2.5. Two alternative outcomes of the scenario have been implemented to satisfy this requirement.

The first change in outcome occurs when the rover fails to obey the policy to halt when it receives new information. In this case it will not receive the detour in time and will crash into the crater.

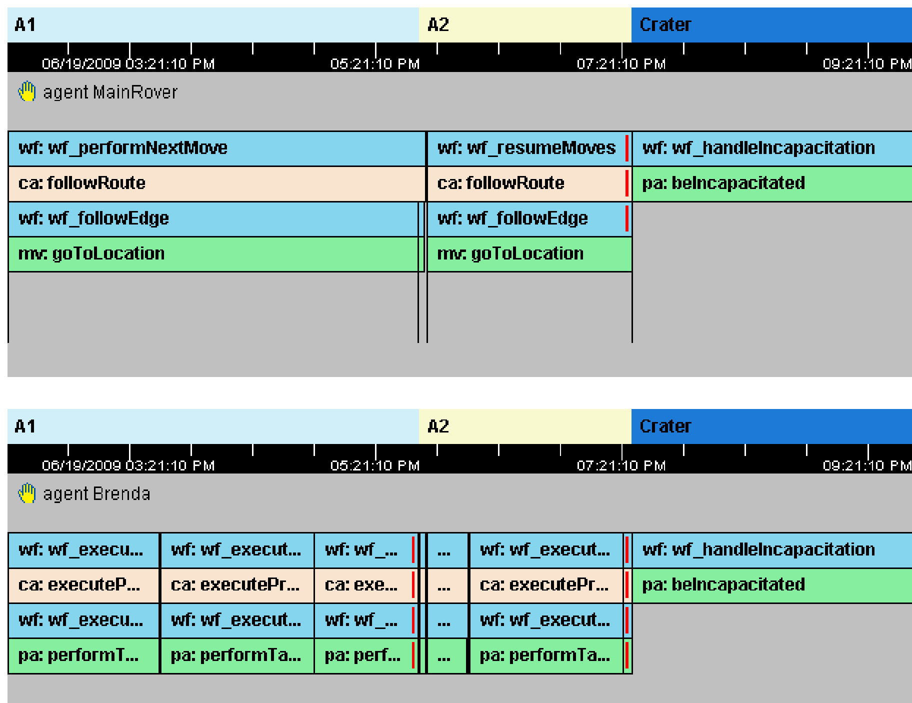

Figure 7.2: Rover driving into crater because Brenda overrides safety policy

The second variation is triggered when the rover disobeys the crater policy and attempts to drive through the crater anyway. This will also result in a crash. However, this variation differs from the first in that the passengers of

the rover are given the option to override a policy decision. The passengers are given this choice because they might have a higher situational awareness. In figure 7.2 the agent viewer timeline is displayed, showing the crash of the rover and Brenda. The rover executes its `wf_resumeMoves` workframe because Brenda did not want to take a detour.

These alternative scenarios are especially interesting when the simulation is used for a human-in-the-loop experiment. In that case the variations offer the human subject tools to influence his environment in ways that will have actual consequences in the rest of the scenario.

7.4 Rover requesting the route

As described in requirement S.A.7 the rover must be able handle routes. Therefore we need to determine the route as per requirement S.A.8. For this we built the `RoutePlanner` which works as follows.

An ePartner can obtain a route by sending a request (containing both the current and desired location of the actor) to the route planner agent. In this simulation the route planner is a stub, capable only of returning the routes as defined in the scenario.

A route consists of a sequence of edges connecting neighbouring waypoints. These waypoints are modelled as `Area` definitions in Brahms. Brahms offers a `Path` concept to model the edges connecting these waypoints, but unfortunately it is impossible to assign attributes other than distance to these default paths. Therefore, agents cannot have beliefs about the depth between the two areas when the `Path` concept is used. To avoid this limitation, the edges are modelled as instances of a custom `Edge` class. This gives the flexibility necessary for modelling the environment.

7.5 Rover receiving new information

To simulate the detection of new information when the rover enters an area, a system consisting of a `SimulationLeader` and a number of `SimulationListeners` is used. We have chosen to use a simulation leader rather than a “real” sensor implementation because we wanted an easy and clear way of influencing the outcome of the simulation. This differs from a real environment, where the rover would be fed information continuously.

During the rover’s movement to its destination, its arrival at waypoint A2 is noticed by a listener agent. This listener then communicates the arrival to the simulation leader. The leader checks whether rovers arriving at A2 should be given new information, and consequently informs the eRover that there is a crater on the way between A2 and the habitat.

Figure 7.4 presents a state diagram of the process that occurs when new potentially hazardous information arrives at the eRover. The eRover will first check if the current route is still authorised. When negative authorisation is received, the rover asks its passengers whether to obey the policy and request a detour. To passengers may override the policy and let the rover continue anyway. If the policy is obeyed, a new route is requested from the route planner and the regular authorisation process starts. Figure 7.3 shows the agent viewer

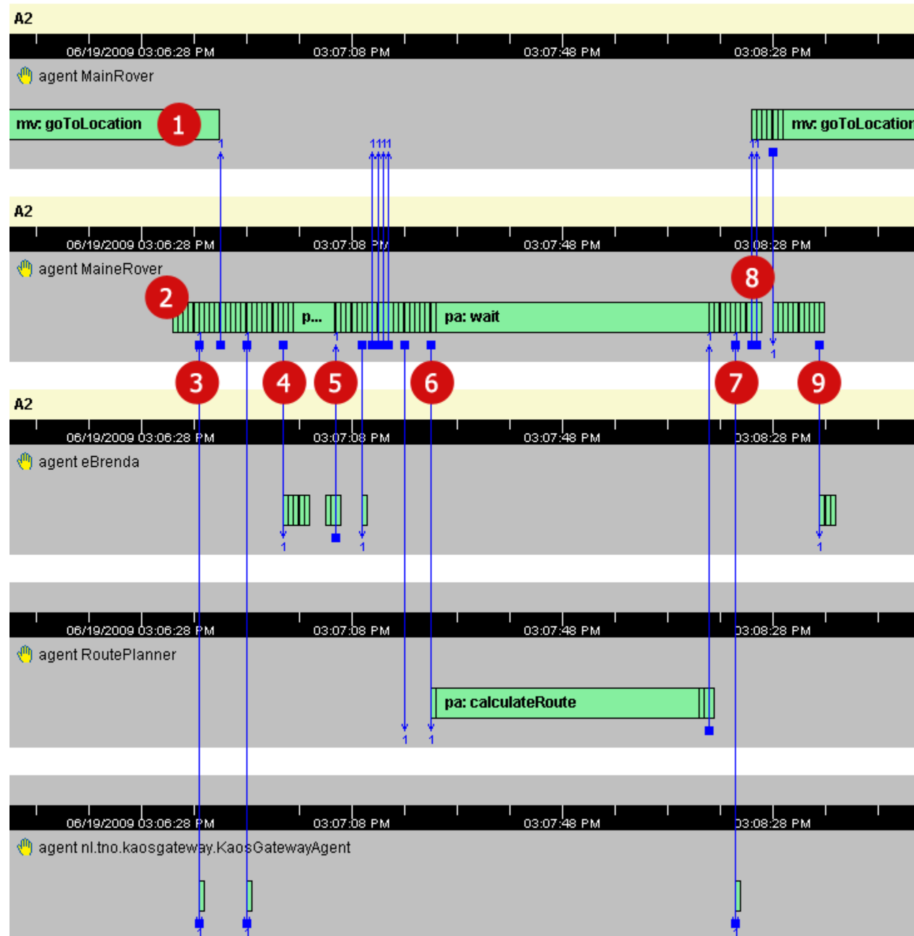

Figure 7.3: Agents collaborating when new information arrives
Rover is moving from waypoint A2 to the crater (1) when new information about the environment arrives at eRover (2). KAoS Gateway Agent informs eRover that it is obligated to halt his rover and tells eRover that the current route is no longer authorised (3). eRover asks the passengers of his rover whether to obey this policy (4). Once eBrenda obeys (5), eRover requests a detour from the route planner (6). eRover requests authorisation to follow this new route (7). Eventually, the rover (8) and Brenda are informed about the detour (9).

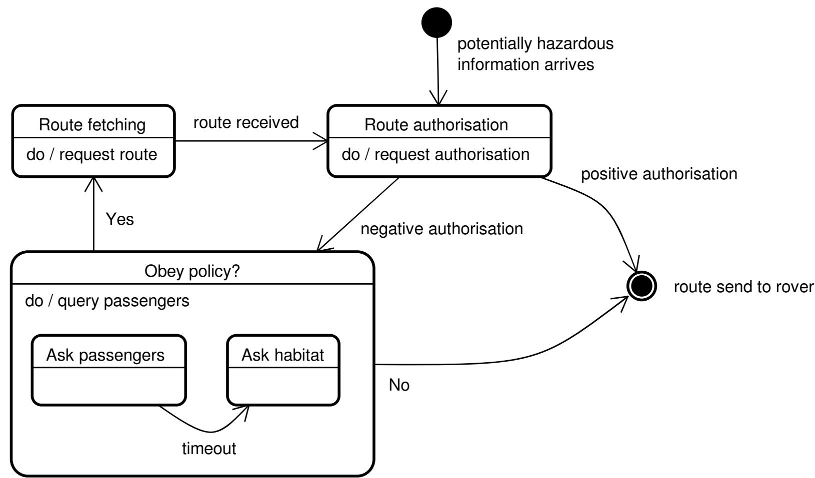

Figure 7.4: State diagram when new information arrives at the eRover

timeline with eRover, rover, eBrenda, route planner and KAoS Gateway Agent. This timeline corresponds to the state diagram from figure 7.4 in case one of the passengers obeys the policy.

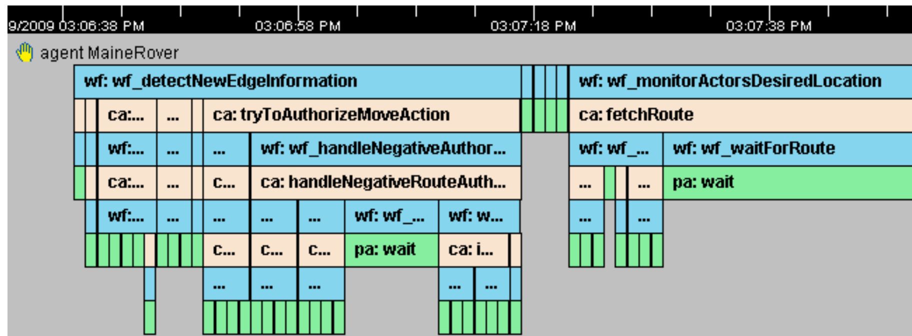

Figure 7.5: eRovers activity stack when new information arrives

In this figure, the interplay between workframes (wf), composite activities (ca) and primitive activities (pa) in Brahms is visible. This excerpt from the agent viewer timeline shows the activity stack when the eRover detects new information. eRover requests authorisation and handles the negative authorisation it receives. Eventually eRover requests a new route.

When the passengers do not respond to the rovers request, the rover forwards it question to the crew members in the habitat. In this scenario, Herman is located at the habitat and will always respond to the rovers request.

7.6 Assisting Benny with his spacesuit failure

Requirements S.A.9 and S.A.11 are concerned with aiding Benny who has become unconscious. The most interesting part is the process in which several steps should be executed by Brenda to put Benny in the “lateral recovery position”. This intricate interplay between eBrenda supplying information to Brenda and other simultaneous messages from the rover asking if it should take a detour lead to an increased cognitive task load of Brenda. The collaboration between Brenda and her ePartner is presented in figure 7.6. Figure 7.7 shows how Brenda is interrupted by the rovers detour.

The collection of steps has been modelled as a Java object called `Procedure`. To complete a `Procedure` Brenda will send an acknowledgement for each step that she completes.

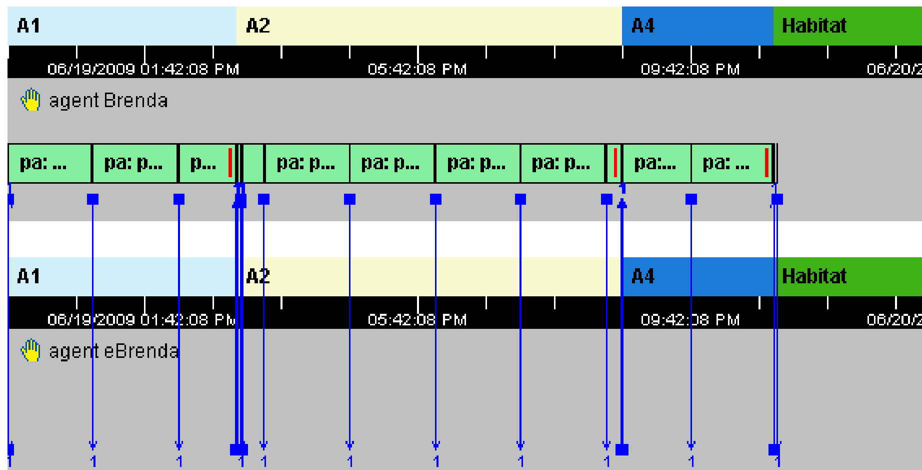

Figure 7.6: Brenda collaborating with her ePartner to help Benny
In this figure, Brenda is executing steps of a procedure to put Benny in the lateral recovery position. Activities are represented as green boxes. She informs eBrenda of the completion of a step before she executes the following step. These inform messages, send as communicative acts, are represented as blue arrows in the agent viewer. Note that Brenda is moving towards the Habitat because she is a passenger on the rover.

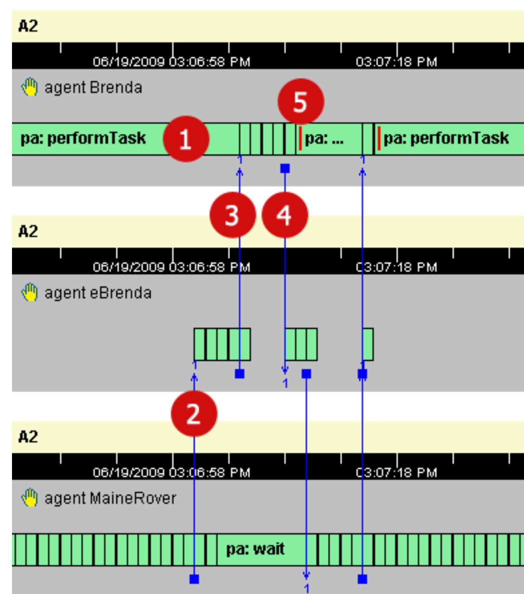

Figure 7.7: eBrenda interrupting Brenda, caused by inform from eRover Brenda is executing a procedure step (1) when eRover informs eBrenda about a detour (2). Brenda halts the procedure for a moment to handle this message (3). After sending her response to eBrenda (4), she resumes her procedure (5). A resumed procedure is represented as a red line at the left side of an activity.

Chapter 8

Human-in-the-loop

Because the development of the KAoS/Brahms bridge and the simulation of the MECA scenario progressed faster than expected, an extension to the original scenario could be made. As mentioned in the preliminaries, many of the requirements in the MECA-project are refined using human-in-the-loop (HITL) experiments. Our supervisor wanted to know whether the Brahms simulation we built could be extended to serve as an underlying simulation driver for such an experiment.

In this chapter we show how the original simulation has been adapted for use in a human-in-the-loop experiment. Section 8.1 identifies the issues that need to addressed for the simulation to be used in experiments. A number of possible solutions for each of these issues is listed in section 8.2. In section 8.3 the implementation of the selected solutions is described.

8.1 Analysis

The main change in the simulation is that it will have to accommodate a human subject as an actor in the scenario. This requires two significant adjustments.

First, the simulation needs to have some kind of interface to be able to run a true HITL experiment. A human subject can use this interface for interaction with the other agents in the simulation. This interface needs to be designed and the best location to insert it in the simulation has to be determined.

Second, for human subjects to interact with a simulation, that simulation has to run in real time. That is, one second in simulated time should correspond to one second in real time. Brahms does not support this by default, so a solution has to be found to prevent the simulation from proceeding too fast for the human subject to keep up.

8.2 Design

Section 8.2.1 discusses three possible locations to insert an interface into the simulation. Three potential solutions to the problem of running the simulation in real time are discussed in section 8.2.2.

8.2.1 Interface

When an agent simulating a human is notified of the occurrence of an event in the simulation, a message is sent along a chain from simulation to that agent, shown in figure 8.1. First, the simulation sends a message to the ePartner of the human in question. The ePartner then forwards this message to the agent it is the ePartner of. This agent finally determines what it wants to be done and sends its response back.

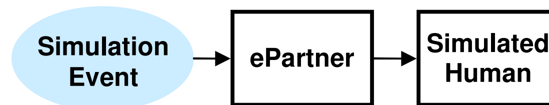

Figure 8.1: Event processing sequence when a human actor is simulated

When the agent simulating the human is to be substituted for an actual human, there are three possible locations in this chain where an interface can be inserted. The first option is to integrate the interface in the ePartner, shown in figure 8.2. The ePartner can then use Java activities to update it.

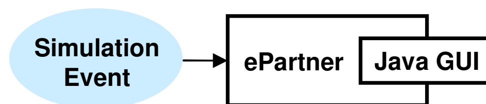

Figure 8.2: Event processing sequence when the interface is attached directly to the ePartner

The advantage of this approach it that it is very easy to realise. However, there are a number of drawbacks. The first problem is that to facilitate the integration of the interface major changes have to be made to ePartner code. These changes go against the clear separation of responsibilities a good design should present.

A second issue is caused by the fact that the ePartner should function well both in simulation and in HITL-mode. This means that all message handling code needs to be implemented twice: once to communicate with the agent simulating the human and once to communicate with the interface an actual human could use.

A second option is shown in figure 8.3. In this case a new agent is created with the sole responsibility of maintaining the interface, allowing the ePartner to remain unchanged. This fixes the ambiguous responsibilities introduced by the first solution.

An issue with this approach stems from the fact that some information about the human actor is relevant regardless of whether its behaviour is simulated or due to an actual human. Examples of such information are the human's membership of a team or the fact that the human is a passenger of the rover. The external agent needs to be a Java agent to efficiently update the interface, so it cannot use Brahms beliefs to represent this information. Because the

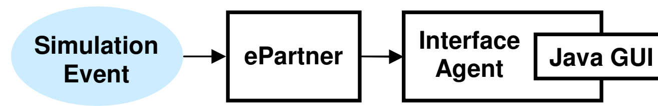

Figure 8.3: Event processing sequence when the interface is attached to a separate agent

simulated agent can only use Brahms beliefs, this again leads to redundant information, making the simulation harder to maintain.

The final option is to keep the separate interface agent suggested by option two, but place it behind the simulated human rather than behind the ePartner, as shown in figure 8.4. This allows the interface agent to use the beliefs of the simulated human agent and retains the other advantages demonstrated by the second option. An added benefit of this approach is that it allows the actions of an agent to be shown in the interface, even if its behaviour is being simulated. However, the simulated human agent needs a slight adaptation to allow the forwarding of messages when an interface is present. Furthermore, the complexity of the system as a whole increases because yet another agent is added to the chain.

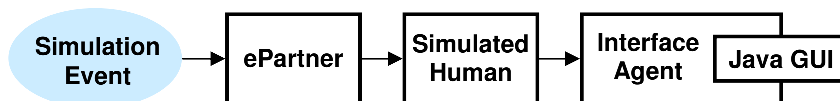

Figure 8.4: Event processing sequence when the interface is attached to a separate agent, leaving simulation code unchanged

Conclusion

With the discussion above in mind, we decide to use the third option: the introduction of an extra agent behind the simulated human agent. It offers the benefit of a clear separation of responsibilities, which greatly facilitates switching between simulation and HITL-mode. Of course, the simulated human will have to be adapted, but this will be a minor change. We also believe the increase in system complexity is relatively small. After all, while this approach introduces a new agent, it does not require maintaining redundant information in separate simulation and HITL versions of the system.

8.2.2 Simulation in real time

When a scenario is developed in Brahms, modellers can specify the duration of activities. However, these durations matter only when a simulation run is inspected after its completion. During the run itself the simulation is executed as fast as the host computer will allow. Because of this, simulating a scenario that covers a few hours may complete in just a few seconds.

The difference between simulated time and execution time causes problems when an actual real-time simulation is needed for a HITL experiment. The execution time therefore has to be slowed down for Brahms to be useful in an experiment.

One way to achieve this is the introduction of *sleep activities*. These are Java activities that invoke the Java `Thread.sleep()` method to pause the thread of execution of an agent for a given number of milliseconds. By invoking such sleep methods whenever the simulation needs to take "real" time, the simulation as a whole slows down to speeds suitable for HITL experiments.

This approach has a number of significant downsides, however. The first issue is that an agent cannot be interrupted in any way while the sleep activity is being executed. For instance, detectables that trigger on the arrival of communicative acts will fail to trigger, which disrupts the execution of protocols.

A second issue is that inserting sleep activities wherever execution needs to be real-time is rather bothersome, as it requires changes in large parts of the simulation. Because the activities themselves will also need to define their durations, this results in unwanted redundancy. This can in turn lead to maintainability problems.

A second way to address the issue of real-time execution is to revise the way in which the scenario is set up. If the scenario is defined to be driven by events rather than by strictly specified time constraints, the need to slow down the simulation is reduced to slowing down the dispatching of these events.

While this solution solves most problems the first one fails to address, there still are some issues that need to be resolved. The most important issue is that the implementation of the scenario given in chapter 7 has not been developed with an event-based architecture in mind. Rewriting the implementation to fit such an architecture would require a lot of work to be redone.

Another issue is that some activities will still need to take time. When the rover moves from one location to the next for example, it should take a while before it arrives. Unfortunately it is exactly during these long-running activities that interrupts are to be expected. Another concern is that for the event-based approach to work, every agent needs to be equipped with event handling code which will, again, cause additional maintenance.

The third and final option is an adaptation of the first. In this approach there is separate agent who manages the time throughout the entire simulation. This is possible due to the way the Brahms VM handles external agents when run in simulation mode.

In simulation mode an agent specifies at what point in simulated time an action will terminate before it start performing it. When the simulation reaches that point, it will not continue until the agent has indeed finished performing the activity. Using this principle, when an agent sleeps for a specified number of (milli-)seconds, the simulation will wait for the agent to wake up before proceeding. This causes regularly spaced delays that slow down the simulation so that humans can interact with it.

Because this can all be done in a single external agent, the responsibility for managing time is nicely isolated. This also facilitates switching between

simulation and HITL-mode.

An issue with this approach is the exact duration of the delays the sleep activity causes. If this duration is too short, the processing time needed to execute the simulation itself becomes a factor and the simulation time will seem too slow compared to the real time. If on the other hand the duration is too long, simulation actions will be executed in bursts. This might cause a human to miss certain events because they fly by too quickly, while at other times the simulation may seem unresponsive for long periods.

Conclusion

Based on the discussion above we decide to use the approach described in the final option, in which a separate agent manages the time. The main reason for this is the fact that it cleanly separates the code dealing with time from the Brahms model and, consequently, that it requires virtually no changes in the model to run. While a usable delay interval needs to be determined, this should not be hard to do with some experimenting.

8.3 Implementation

In this section the implementation of the graphical user interface (GUI) is described. To make the system work as described in section 8.2.1 an external agent called the *GUI Agent* is introduced. This agent forms the link between a Brahms agent and the Java GUI. Every simulated human that needs to be replaced will have a `hasInterface` relationship with its GUI Agent. To control the GUI Agent, all communicative acts that the simulated human receives are handled by a new high priority workframe. This workframe forwards all messages to the GUI agent, which will translate them into messages for the GUI.

The right hand side of the screenshot in figure 8.5 displays the current step in the procedure that Brenda should perform. Below this procedure, an abstract map of the environment is displayed with information about the locations of actors, waypoints and the current route of the rover (shown as a red line). The hand left side of the screen shows the messages from the ePartner to Brenda. Brenda has to tap the `OK` button of each message to acknowledge the information to her ePartner.

The screenshot in figure 8.6 shows a prompt from the rover, asking Brenda whether or not a policy should be obeyed. Brenda should respond to this prompt before it times out, as stated in section 7.5. This prompt enables Brenda to influence the scenario.

As stated in section 8.2.1, this solution facilitates switching between simulated and HITL mode. Furthermore, a combination of these two modes is offered, by running the scenario in simulated mode while displaying the GUI. This allows the analysis of the behaviour of the simulated humans by observing their actions on their GUI. Configuration files are provided to specify what combination of simulation and HITL mode should be used.

Requirement G.B.4 of appendix A.3 specifies that the interface should be usable on a touch screen. Therefore the GUI uses large text and icons to enable a human to operate the system by tapping his finger on the screen.

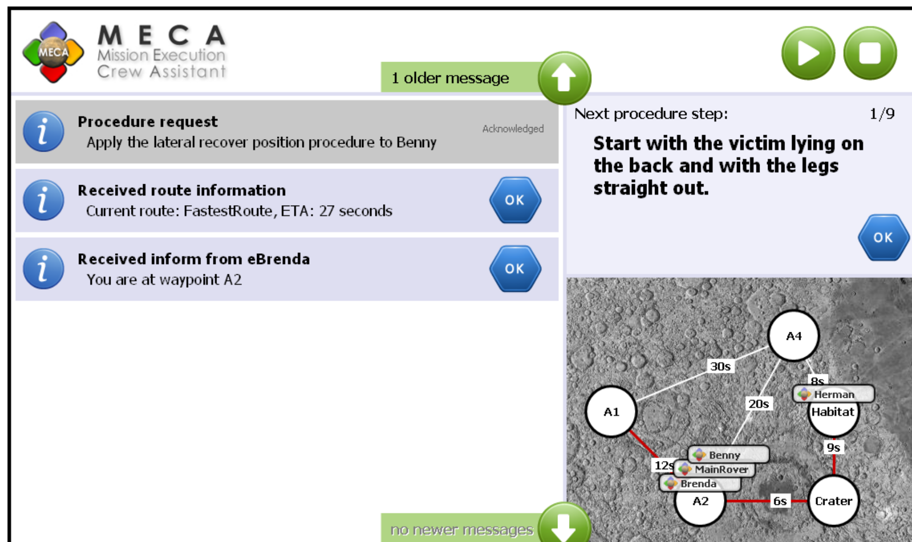

Figure 8.5: Touch screen user interface for Brenda
Left: information that needs to be acknowledged
Upper right: a step from the active procedure
Bottom right: a map showing actor locations and waypoints in the simulation

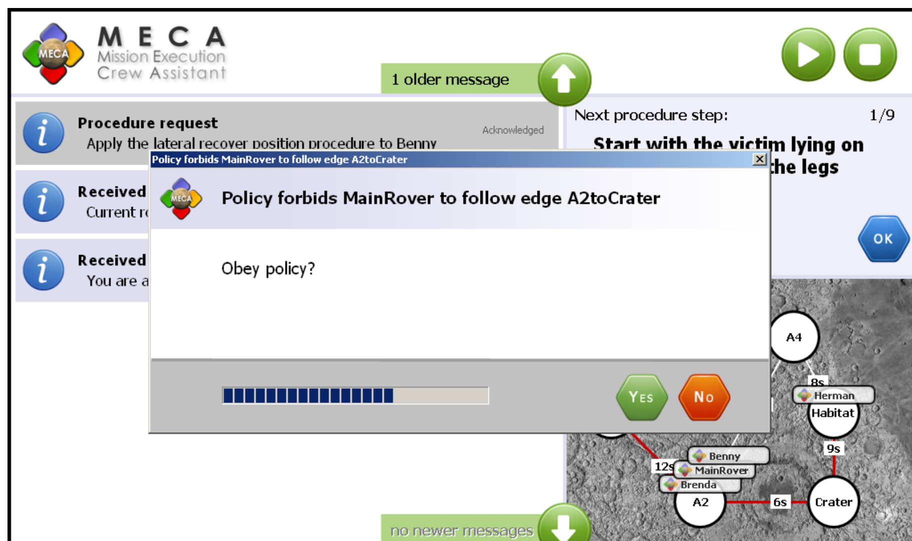

Figure 8.6: Prompt from rover on touch screen user interface

Part IV

Conclusions, Recommendations and Future work

Chapter 9

Conclusions

Our conclusions are split into two categories. In section 9.1 the work we delivered is described. Section 9.2 discusses tentative results and possible applications of the components we built. These applications themselves will have to be subjected to further research.

9.1 Work delivered

Our work has resulted in three main deliverables: the KAoS/Brahms bridge, the simulation of a MECA scenario and the modification of that scenario for a human-in-the-loop experiment, which includes a graphical user interface. These will be further discussed in this section. An analysis of the fulfilment of the requirements for these three components is included in appendix C.

By using the KAoS/Brahms bridge, Brahms multi-agent systems are now able to query KAoS policies. The part of the bridge implemented in Brahms itself, the Brahms Policy Library, is independent of KAoS. The Policy Library can therefore be re-used for the integration of other policy services, should they be needed.

Because it requires virtually no changes in existing model code, the bridge can be easily integrated into existing Brahms projects. The only changes required are those necessary to handle policies, which would be needed anyway.

The system also safeguards the autonomy of agents by allowing them to disobey policies if they deem it necessary. This preserves the notion that agents should be autonomous, and allows human participants in future experiments to override decisions dictated by the policy system.

The Brahms simulation of the scenario (described in appendix B.2) uses the KAoS/Brahms bridge to query KAoS for policy decisions. Because these can either be obeyed or ignored, alternative scenarios emerge that may result in different outcomes.

Finally, we extended the simulation to make it suitable for a human-in-the-loop experiment. A graphical user interface was built that allows a human subject to interact directly with any ePartner. The user interface and the simulation may be used in an actual human-in-the-loop experiment in Fall 2009.

Because of the modular approach taken in the architecture of the simulation, the GUI and the KAoS/Brahms bridge, each of these can be replaced by other implementations with little effort. The GUI can be used as a front-end to a different simulation and the simulation can run with a different GUI or even without any GUI at all. The KAoS/Brahms bridge can be integrated in any other Brahms project, and the Brahms Policy Library can be easily extended with support for other policy services. Even a Wizard of Oz "service" would be possible, in which the experiment leader determines if a policy should fire or not.

9.2 Tentative results

A first conclusion to draw is that Brahms and KAoS can indeed be integrated, as demonstrated by the KAoS/Brahms bridge described in part II. This bridge has been successfully used in the implementation of a scenario (part III) and during the creation of a demonstration project.

The ability to use the KAoS/Brahms bridge in the demonstration project demonstrates its independence of the MECA project. The Brahms Policy Library itself is even independent of KAoS, as it only sends general requests to the KAoS-specific gateway agent. While it has not been demonstrated in this report, we expect no large problems if KAoS were to be substituted for another policy service. After all, an implementation of a gateway agent for that new service is all the Policy Library needs. Because of the abstraction offered by the Policy Library, agents in the simulation would not even notice that anything has changed if the policy service is replaced.

However, while the bridge has sufficient functionality to be of use in experiments, there are still details that need closer attention if the entire spectrum of KAoS policies is to be supported. The bridge currently supports simple authorisations and obligations, but more work is needed in both the Policy Library and the KAoS Gateway Agent to unlock the finely-grained options KAoS offers. Suggestions for possible extensions are given in chapter 11.

A more general conclusion is that using Brahms and KAoS it is possible to implement a scenario which relies on the principles of both work practice and policies. In our experience, this resulted in a richer modelling experience: we feel that the combination of the bottom-up style used in Brahms' work practice approach with the top-down rules specified by KAoS policies results in a more natural modelling style.

If this combination is further extended to include human subjects, we expect new opportunities to emerge that combine the actual human experience of the subject with the benefits of simulated agents. The simple graphical user interface discussed in the previous section shows that it is at the very least possible to set up a human-in-the-loop experiment using this approach. How the results of this experiment compare to other human-in-the-loop approaches needs to be determined by future research.

Chapter 10

Recommendations

In the course of our project we worked intensively with Brahms and KAoS. In this chapter, we present several recommendations for Brahms and KAoS based on this experience.

10.1 Recommendations for Brahms

Abstraction in activities

Brahms allows agents to be member of different agent groups, allowing them to use the activities and workframes defined by the groups they are member of. Activities can also be overridden when a subgroup or agent needs a more specific implementation. It is not possible however to use the Template Method pattern: state that a subgroup or agent should implement an activity without giving an implementation at all. Java accomplishes this by allowing methods to be declared abstract. Because Template Methods are not supported, is it difficult to create "pluggable" libraries, which provide a general approach that can be adapted to specific circumstances.

A possible solution to this issue would be to allow groups to define abstract activities. These would have to be implemented by group members or subgroups, or cause a compile error if they fail to provide an implementation.

Related to this issue is the fact that Brahms does not provide a construct to invoke the overridden method from within the overriding method, like the "super" call in Java or Python. This can be addressed with the introduction of a "super" construct. Because the multiple inheritance architecture of Brahms might cause problems with this construct, an approach similar to that in Python might be used, There, the overriding method specifies to which group the super call should go, avoiding problems when multiple supergroups define methods with identical names.

Abstraction in classes

The `PolicyAction` class in the KAoS/Brahms bridge serves as base class which modellers can extend to communicate about action that involve policies. An instance of the Policy Action class itself however would not be useful. It provides

information needed for authorisation and obligations, but does not represent any executable action: it is an abstract concept.

It would be useful to have some keyword designating a class to be "non-instantiatable", such as the "abstract" keyword for Java classes. A simpler version of the same idea is the possibility of defining interfaces in Brahms, which merely specify a contract that the inheriting class needs to follow. These two additions would facilitate the creation of highly re-usable components in Brahms, such as libraries.

Detectables on bound variables

Brahms provides the concept of *detectables* to enable the interruption of workframes when an agent detects changes in its beliefs. However, it is currently not possible to use detectables alone for highly specific interruptions, such as the arrival of a Communicative Act in answer to a specific Act sent earlier. This is not possible because detectables can only fire on a single attribute or relation, not a set of conditions.

Currently thoughtframes are used to resolve this issue: the thoughtframe fires on a set of conditions and sets an attribute, which in turn triggers the detectable to interrupt the execution of the workframe. However, this introduces a lot of overhead.

It also violates the principle that code related to a single task should be kept close together. To determine why a detectable did or did not fire it is currently necessary to look up the attribute and then look up the thoughtframe setting this attribute. It would be easier if a detectable could be set to fire on a set of conditions, using a variable binding scheme similar to the one available for workframes.

Running the Composer's Brahms VM in a separate Java VM

When Brahms models are run directly from within the Composer, the executing Brahms VM is running in the Java VM instance of the Composer itself. This causes problems when the execution of a model generates serious errors.

A striking example of such an error is when the model (or a library used in an external agent) invokes `System.exit()`, terminating the entire Java VM. Because the Composer is running in the same Java VM as the model, this also causes the Compose to crash "out of nowhere", with no option to save or recover any possibly unsaved work.

Another disadvantage of using the same Java VM for both the Composer and the Brahms VM emerges when the model relies on external Java libraries. When these libraries have been loaded into the JVM, they remain there for the rest of the run, i.e. until the Composer is shut down. This means that the Composer has to be restarted every time the libraries change, which is particularly bothersome when developing external Java agents or activities.

A solution to both these problem would be to start the Brahms VM running a model in a separate Java VM, much like eclipse does. That way, while errors may still cause the model to crash, at least the Composer will remain stable and running. Also, libraries are loaded anew each time a new Java VM is started, speeding up development for external agents and activities.

10.2 Recommendations for KAoS

Define policies over agent instances in an ontology

OWL ontologies allow the definition of instances of actors [5], so it should be possible to define policies for those actors. Therefore it should be possible to define a policy for an actor before this actor is registered in the KAoS Directory Service. Currently policies can only be defined for actor classes, registered agents and virtual agents. While OWL supports policies for actors that are not registered, KAoS does not. We therefore propose that KAoS is extended to include this capability, to conform with the OWL specifications.

Obligations over the instigator of the obligation

KAoS only supports obligations of the form "Before an agent of class X does Y, every agent of class X is obligated to do Z". This means that whenever an instance of class X triggers this obligation, *every* agent of class X is obligated to perform Z.

However there are situations in which only the specific agent that *triggers* the obligation should be obligated to do Z, while the obligation should still hold for every agent of class X. It is possible to define this obligation for all the instances of class X individually, but this introduces unwanted replication. We therefore propose that support is added to create obligations of the form "Before an agent A of class X does Y, A is obligated to do Z".

Chapter 11

Future work

This chapter describes future work and improvements that can be made to the system we have delivered. It is split up in different parts. Future work regarding the KAoS/Brahms bridge, the Ontology Builder and the MECA simulation can be found sections 11.1, 11.2 and 11.3 respectively.

11.1 KAoS/Brahms bridge

Obligations before and after actions

KAoS has the ability to differentiate between obligations that must be fulfilled before and those that must be fulfilled after performing an action. The KAoS/Brahms bridge does not currently communicate this difference to the Brahms agent: all obligations are assumed to be before-obligations. Future editions of the bridge may support this difference, for example by adding an extra attribute to a Policy Action representing an obligation. This attribute could then be used by the application modeller to detect the difference between the two types and act accordingly.

Obligations involving different actors

An agent can be obligated to perform an action when another agent does something. For example, a pool attendant is obligated to shout to a child who is running over a wet floor. KAoS can be used to define this policy, but the bridge is currently not able to inform the pool attendant of his obligation to shout when it receives the authorisation request from the child.

This is caused by the fact that the Policy Library currently does not include functionality to handle the interruption of the normal execution such an externally triggered obligation would cause. What is more, the KAoS/Brahms bridge itself does not have the ability to take initiative to contact an agent and inform it of a new obligation. It can only respond to queries the agent makes himself.

Obligations with object properties

In KAoS it is possible to define obligations in which restrictions are placed on attributes with an object value. Say for example that a speaker object has an attribute specifying the location of the speaker on the rover. In that case an obligation with an object value would be that a rover must execute a beep action on a speaker when moving backward, and that this speaker must be on the rear of the rover.

The restriction on the speaker location can as of yet not be communicated using the bridge. This is caused by the lack of functionality in the bridge to transform the nested restrictions from KAoS' XML format into a Brahms Restriction object.

Authorisations based on nested attributes

Highly similar to the previous issue, it is currently not possible to enforce policies that determine authorisation based on "nested attributes": attributes of objects that are themselves attributes of an action. For example, the following policy can not be enforced by the bridge: "A rover is not authorised to follow an edge if the end node of that edge lies higher than 500m". Here the attribute "height" of a node is a nested attribute.

This policy is not enforceable because the bridge does not translate all nested objects when it converts a Brahms Policy Action into a KAoS Action Instance Description (AID). Instead, it simply replaces those objects with their names. A possible way to solve this would be to recursively translate the Brahms object and its attributes into some sort of nested AID. However, this mapping is not straightforward because AIDs cannot be nested.

Simulate time necessary for authorisation lookup

Because the simulation blocks while the KAoS/Brahms bridge contacts KAoS, the result of a policy lookup is instantaneously known by the requesting agent in simulation time. It would make simulations more realistic if a developer could specify some unit of time needed to complete a lookup.

Policy Library utility methods

Even though the Policy Library facilitates a wide range of possible interactions with policy service providers, the current API is somewhat spartan. To facilitate a faster development process, it would be useful to integrate more utility methods in the library.

One such utility method is a non-blocking method to retrieve policy decisions. Currently an agent is forced to sit around and do nothing until the policy library returns with the answer to his request. Introducing a non-blocking call would allow the agent to use that waiting time to perform other useful actions.

11.2 Ontology Builder

Exporting core Brahms concepts

The Ontology Builder currently only exports the ontology of the concepts that are defined in a Brahms model itself. However, these models rely on a number of core Brahms concepts which are not exported, such as "Area" or "Active Instance". While the model ontologies can be used without problems, exporting the ontology of the core concepts as well would enable policy makers to define even broader policies. An example would be a policy that applies to *any* agent (Active Instance) in the simulation.

11.3 MECA Simulation

The simulation of the MECA scenario in conjunction with the ePartner GUI is ready for use in a human-in-the-loop experiment. Based on these experiments refinements to both the simulation and GUI can be made. Eventually these experiments should lead to a refined requirements baseline for the MECA project.

Synthetic task environment

To enhance the reality of the simulation for the human subject, a synthetic task environment (STE) could be created in which the simulation is visualised. This STE could be a virtual reality environment based on the Unreal Tournament game-engine that displays the actions of all actors. To construct such a system, a method has to be designed which allows the Brahms simulation to communicate with the game-engine.

Part V

Appendices

Appendix A

Requirements analysis document

This appendix lists the requirements for the MECA simulation, the KAoS/Brahms bridge and the ePartner's Graphical User Interface in sections A.1, A.2 and A.3 respectively. An overview of the fulfilment of these requirements is given in appendix C.

Requirements are prioritised using the MoSCoW model, which introduces four priority levels. The highest priority class contains work that MUST be done for the solution to be acceptable by the client. A slightly lower priority is assigned to work that SHOULD be done to implement important but non-vital functionality. This is followed by work that COULD be done if the implementation follows naturally from other features that had to be implemented anyway. Finally the lowest priority is given to work that the client WOULD like to have done if time remains when all features with higher priority have been completed.

A.1 MECA simulation

This section describes requirements for the MECA simulation, which is a Brahms model capable of simulating the scenario's in appendix B. Requirements are defined for each individual actor in this simulation.

S.A. Functional requirements

- S.A.1. The "first iteration" scenario from appendix B.1 MUST be simulated in Brahms, containing the actors: eRover, Rover, RoutePlanner and PolicyChecker.
- S.A.2. The "second iteration" scenario from appendix B.2 SHOULD be simulated in Brahms, containing the actors: eRover, Rover, eBrenda, Brenda, eBenny, Benny, RoutePlanner and PolicyChecker.
- S.A.3. The simulation SHOULD stop in case one of the agents arrives at the crater.
- S.A.4. The simulation SHOULD indicate which agents crashed into the crater.
- S.A.5. The eRover agent MUST be able to
  - S.A.5.1. understand when he has to check policies,

- S.A.5.2. comply with policies (i.e. choose an alternative route, which avoids certain inaccessible areas),
- S.A.5.3. request a route from the RoutePlanner agent,
- S.A.5.4. tell the Rover to drive to a specific location and
- S.A.5.5. handle new information regarding depth of the terrain.

S.A.6. The eRover agent SHOULD be able to

- S.A.6.1. communicate the rover's ETA to other ePartners.
- S.A.6.2. communicate information about decisions it makes to other ePartners.
- S.A.6.3. be obligated to halt the Rover when potentially hazardous information arrives.

S.A.7. The Rover agent MUST be able to

- S.A.7.1. follow routes provided by the eRover,
- S.A.7.2. calculate and display ETA and travel time on arrival at a waypoint,
- S.A.7.3. notify the eRover of arrival at the end of the route and
- S.A.7.4. give a notification on arrival.

S.A.8. The RoutePlanner agent MUST be able to provide[1] requesting agents with routes between two geography nodes. The request includes a list of nodes that must be excluded from the route.

S.A.9. Benny SHOULD not respond to any communication from other agents when he is unconscious.

S.A.10. Brenda SHOULD be able to acknowledge the information the ePartner sends her.

S.A.11. eBrenda SHOULD be able to

- S.A.11.1. display the ETA of the Rover to Brenda.
- S.A.11.2. communicate Benny's vital signs to Brenda.
- S.A.11.3. give the assignment to put Benny in the "lateral recovery position".
- S.A.11.4. show the steps of the "lateral recovery position" to Brenda.
- S.A.11.5. pause the guide on "lateral recovery position" if other messages arrive, e.g. the rover deciding to take a detour.
- S.A.11.6. receive the acknowledgement on information given to Brenda.

S.B. Non-functional requirements

S.B.1. Testing

- S.B.1.1. A regression test exercising the MECA simulation COULD be included in the set of tests.

S.B.2. Documentation

- S.B.2.1. Documentation SHOULD be delivered on how to run the simulation for inexperienced users.

[1] Note that due to the use of "provide" instead of "calculate", it is not required to actually execute a path finding algorithm. The route may be pre-computed or even hard-coded.

S.C. Constraints

S.C.1. The host machine must have to following software installed:

S.C.1.1. Java version 1.6 or higher,

S.C.1.2. KAoS version 2.0 or higher,

S.C.1.3. Brahms version 1.3.0 or higher and

S.C.1.4. KAoS/Brahms bridge 0.5 or higher.

A.2 KAoS/Brahms bridge

This section describes requirements for the KAoS/Brahms bridge, which provides agents in the MECA simulation (section A.1) with the ability to query policies in KAoS.

B.A. Functional requirements

B.A.1. The KAoS/Brahms bridge

B.A.1.1. MUST be able to support the MECA simulation described in section A.1.

B.A.1.2. SHOULD be generic; it should not be specific to the MECA simulation.

B.A.1.3. MUST translate policy requests into a corresponding KAoS AID.

B.A.1.4. MUST relay AID objects to the KAoS Policy Service.

B.A.1.5. MUST receive responses from KAoS and transform them to a format understandable for Brahms agents.

B.A.1.6. MUST send the response for the policy requests to the requesting agent in the Brahms environment.

B.A.1.7. MUST be able to handle negative authorisation from KAoS.

B.A.1.8. SHOULD be able to handle obligations from KAoS.

B.A.1.9. SHOULD be able to transform obligations to a format understandable for Brahms agents.

B.A.1.10. SHOULD inform a Brahms agent that sends a policy request about his obligations.

B.A.1.11. COULD provide a way of transforming a Brahms model into an OWL ontology.

B.B. Non-functional requirements

B.B.1. Testing

B.B.1.1. All Java code SHOULD be accompanied by unit-tests.

B.B.1.2. The tests SHOULD achieve 100% statement coverage, excluding exception handling for Brahms exceptions.

B.B.1.3. An integration test exercising both a Brahms model and the KAoS/Brahms bridge simultaneously COULD be included in the set of test.

B.B.2. Documentation

B.B.2.1. All Java interfaces in the KAoS/Brahms bridge SHOULD have JavaDoc documentation for both:

B.B.2.1.1. the general purpose of the interface and

B.B.2.1.2. the purpose and parameters of each method.

B.B.2.2. Documentation SHOULD be delivered on how to integrate the KAoS/Brahms bridge into Brahms projects for people with a basic understanding of Brahms.

B.C. Constraints

B.C.1. The host machine must have to following software installed:

B.C.1.1. Java version 1.6 or higher,

B.C.1.2. KAoS version 2.0 or higher,

B.C.1.3. Brahms version 1.3.0 or higher.

A.3 Graphical User Interface

This section describes requirements for the Graphical User Interface (GUI), which provides Brenda with an interface to her ePartner. In this section Brenda is a human actor in a human-in-the-loop experiment. This ePartner and all other actors run in the MECA simulation described in section A.1.

G.A. Functional requirements

G.A.1. The ePartner GUI

G.A.1.1. MUST provide Brenda with a option to start the simulation.

G.A.1.2. MUST display text messages and prompts from the ePartner.

G.A.1.3. MUST provide Brenda with an option to acknowledge a text message.

G.A.1.4. MUST provide Brenda with an option to answer *yes* or *no* to a prompt.

G.A.1.5. MUST be able to show Brenda a procedure.

G.A.1.6. MUST provide Brenda the option to mark steps of this procedure as finished.

G.A.1.7. SHOULD display the remaining time before a message times out.

G.A.1.8. SHOULD provide Brenda the option to view the map, procedure or overview in case one of them is not on the display.

G.A.1.9. COULD provide a view containing

G.A.1.9.1. a map of the environment,

G.A.1.9.2. the location of all actors,

G.A.1.9.3. the route of the rover and

G.A.1.9.4. ETA of the rover.

G.A.2. The Brahms simulation

G.A.2.1. MUST be able to run in real-time.

G.A.2.2. MUST wait until Brenda has started the simulation.

G.A.2.3. MUST relay messages from eBrenda to the ePartner GUI.

G.A.2.4. SHOULD let the eRover wait until one of the passengers has acknowledged a detour.

G.A.2.5. SHOULD let the actors crash in case one of the passengers does not want to take a detour.

G.A.2.6. SHOULD handle a prompt time out from Brenda, by either

- redirecting the prompt to Herman or
- simulating the redirection to Herman.

G.A.2.7. SHOULD inform Brenda of the result of the redirected prompt.

G.A.3. The task environment

G.A.3.1. COULD offer Brenda a GUI to execute steps of a procedure.

G.A.3.2. WOULD display a model of Benny and functionality to place him in the lateral recovery position.

G.B. Non-functional requirements

G.B.1. The Brahms and KAoS environment MUST be started using a well specified process.

G.B.2. The ePartner GUI MUST run on the same PC as the Brahms simulation.

G.B.3. The ePartner GUI SHOULD work on a screen resolution of 1024×600.

G.B.4. The ePartner GUI SHOULD work on a touch screen.

Appendix B

Scenarios

The development of the MECA scenario is split up in two iterations. The scenario in section B.1 focuses on a rover driving to the habitat. In section B.2, this scenario is expanded with human actors who are assisted by their ePartners.

B.1 First iteration scenario

Setting	The Rover is going to drive to the habitat	
Policy rules	Negative authorisation for Rover to go through crater that is more than 2 meters deep	
Locations	As shown in figure B.1	
Agents	eRover, Rover, RoutePlanner, PolicyChecker	
eRover	Inputs	• Policy rules • Route • Geographic information (crater)
	Outputs	• ETA • Travel time • Route
Route planner	Inputs	• Request for route
	Outputs	• Route
Rover	Inputs	• Route
	Outputs	• Arrival at habitat
Policy checker	Inputs	• Policy request objects
	Outputs	• Authorisation values

Scenario

1. eRover asks the route planner for a route to the Habitat
2. Route planner calculates fastest route
3. Route planner gives route information to eRover
4. eRover checks policies
5. eRover gives route to Rover
6. eRover displays ETA and travel time during whole scenario (for instance: ETA: 9.42 pm, travel time: 10 minutes)
7. Rover starts driving to Habitat
8. eRover receives message about crater between A2 and Habitat
9. eRover checks policies
10. eRover asks RoutePlanner for a new route
11. RoutePlanner calculates detour
12. RoutePlanner communicates detour to eRover
13. eRover communicates detour to Rover
14. Rover takes detour
15. Rover arrives at Habitat
16. Rover reports arrival to eRover
17. eRover shows arrival on map

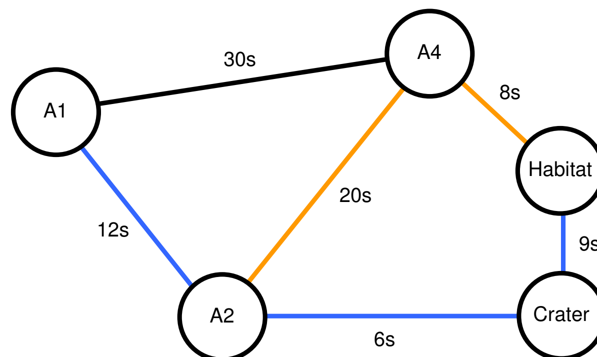

Figure B.1: Map of the environment used in the scenarios. The rover starts at waypoint A1 and drives to the habitat. The fastest route is indicated with a blue line. At A2, the rover receives a detour, indicated with an orange line.

B.2 Second iteration scenario

Setting	• Brenda is in the Rover • Benny is hypothermic and has fainted • The Rover starts driving to the habitat • Brenda is watching Benny and the route of the Rover	
Policy rules	• Negative authorisation for Rover to go through crater that is more than 2 meters deep • Obligation for eRover to halt the Rover when potentially hazardous information arrives	
Locations	As shown in figure B.1	
Agents	eRover, Rover, eBrenda, Brenda, eBenny, Benny, RoutePlanner, PolicyChecker	
eRover	Inputs	• Policy rules • Route • Geographic information (crater)
	Outputs	• ETA • Travel time • Route
Route planner	Inputs	• Request for route
	Outputs	• Route
Rover	Inputs	• Route
	Outputs	• Arrival at habitat
Brenda	Inputs	• Lateral recovery position procedure • Messages and prompts from eBrenda
	Outputs	• *None*
eBrenda	Inputs	• Status updates from eRover
	Outputs	• Lateral recovery position procedure • Messages and prompts to eBrenda
Benny	Inputs	• *None* (unconscious)
	Outputs	• *None* (unconscious)
Policy checker	Inputs	• Policy request objects
	Outputs	• Authorisation values

Scenario	1. eRover asks the route planner for a route to the Habitat 2. Route planner calculates fastest route 3. Route planner gives route information to eRover 4. eRover checks policies 5. eRover gives route to Rover 6. eRover displays travel time during whole scenario (for instance: ETA: 10 minutes) 7. Rover starts driving to habitat 8. eRover gives ETA and travel time to eBrenda 9. eBrenda communicates ETA and travel time to Brenda 10. Brenda acknowledges event 11. eBrenda tells Brenda that Benny is unconscious, breathing and has a heartbeat and that she has to put Benny in lateral recovery position 12. Brenda acknowledges that she will put Benny in lateral recovery position 13. eBrenda shows procedural steps of putting a person in lateral recovery position 14. eRover receives message about crater between waypoint A2 and Habitat 15. eRover checks policies 16. eRover asks RoutePlanner for new route 17. RoutePlanner calculates detour 18. RoutePlanner communicates detour to eRover 19. eRover communicates detour to Rover 20. eRover communicates detour and reason to eBrenda 21. eBrenda pauses lateral recovery position procedure 22. eBrenda reports detour and reason of detour to Brenda 23. Brenda acknowledges event and continues with lateral recovery position procedure 24. Rover takes detour 25. Rover arrives at Habitat 26. Rover reports arrival to eRover 27. eRover shows arrival on map

Appendix C

Requirements evaluation

This appendix contains an overview of the fulfilment of all requirements specified by the supervisor. They have been separated into three different tables corresponding to the three sets of requirements as specified in appendix A.

In the "Completion" column, a "V" indicates that the requirement was fully met and an "X" indicates that it has not been implemented. Where the fulfilment could not be sufficiently indicated with these symbols, the level of completion is explicitly described.

Requirement	Priority	Completion	Relevant sections
S.A.1	Must	V	7
S.A.2	Should	V	7
S.A.3	Should	V	7.3
S.A.4	Should	V	-
S.A.5	Must	V	7.4, 7.5
S.A.6	Should	V	7.5
S.A.7	Must	V	7.4
S.A.8	Must	V	7.4
S.A.9	Should	V	7.6
S.A.10	Should	V	7.6
S.A.11	Should	V	7.6
S.B.1	could	V	5.3
S.B.2	should	V	D.2

Table C.1: MECA Simulation requirements evaluation

Requirement	Priority	Completion	Relevant sections
B.A.1.1	MUST	V	7.2
B.A.1.2	SHOULD	V	3.2, 4.2.2, 6.4.2
B.A.1.3	MUST	V	4.3, 6.2.1
B.A.1.4	MUST	V	4.3, 6.2.1
B.A.1.5	MUST	V	6.2.1
B.A.1.6	MUST	V	6.2.1
B.A.1.7	MUST	V	6.2.1
B.A.1.8	SHOULD	V	6.2.1
B.A.1.9	SHOULD	V	6.2.1
B.A.1.10	SHOULD	V	6.2.1
B.A.1.11	COULD	V	4.2.1, 6.3
B.B.1.1	SHOULD	Core only	5.1, 6.4.1
B.B.1.2	SHOULD	V	5.1, 6.4.1
B.B.1.3	COULD	V	5.2, 6.4.2
B.B.2.1	SHOULD	V	-
B.B.2.2	SHOULD	V	6.1.1, D.3

Table C.2: KAoS/Brahms bridge requirements evaluation

Requirement	Priority	Completion	Relevant sections
G.A.1.1	Must	V	8.3
G.A.1.2	Must	V	8.3
G.A.1.3	Must	V	8.3
G.A.1.4	Must	V	8.3
G.A.1.5	Must	V	8.3
G.A.1.6	Must	V	8.3
G.A.1.7	Must	V	8.3
G.A.1.8	Should	V	8.3
G.A.1.9	Could	V	8.3
G.A.2.1	Must	V	8.2.2
G.A.2.2	Must	V	-
G.A.2.3	Must	V	8.2.1
G.A.2.4	Should	V	7.5, 8.3
G.A.2.5	Should	V	7.5, 8.3
G.A.2.6	Should	V	7.5, 8.3
G.A.2.7	Should	V	8.3
G.A.3.1	Could	X	
G.A.3.2	Would	X	
G.B.1	Must	V	D.2.2
G.B.2	Must	V	8.3, D.2.2
G.B.3	Should	V	8.3
G.B.4	Should	V	8.3

Table C.3: Graphical User Interface requirements evaluation

Appendix D

Installation guide

Before running the MECA simulation the KAoS/Brahms bridge has to be installed. This appendix describes how this is done. Note that all file names in this installation guide are relative to the base of the distribution zip file.

D.1 Installation of the KAoS/Brahms bridge

Make sure the following software is installed before installing the KAoS/Brahms Bridge:

- Java version 1.6 or higher (`http://java.sun.com/javase/downloads/`)
- KAoS version 2.0 or higher (`http://ontology.ihmc.us/`)
- Brahms version 1.3.0 or higher (`http://www.agentisolutions.com/download/`)

Note that a valid Brahms license file is needed to run Brahms models and that access to KAoS downloads was restricted at the moment of writing.

To use the KAoS/Brahms bridge in your models, two files need to be moved to the Brahms installation:

1. Copy `kaosbrahmsbridge/KAoS Gateway Agent/kaosgateway.jar` to your Brahms `deploy` folder.
2. Copy `kaosbrahmsbridge/KAoS Gateway Agent/kaosgateway.properties` to your Brahms `config` folder.
3. Set the `kaos.basedir` property in this file to the base directory of your KAoS installation.

Mac OS X

Some additional steps are needed to set up the bridge in Mac OS X.

1. Open a terminal and execute the command "`cd ~/.MacOSX/`". The folder name is case sensitive, create the folder if it does not exist yet.
2. Execute the command "`vim environment.plist`".

3. Store the following in the file or add the new key if it already exists. If necessary, adjust the path between the `<string>` tags to the path of your Brahms Agent Environment directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>BRAHMS_HOME</key>
    <string>/Applications/Brahms/AgentEnvironment</string>
  </dict>
</plist>
```

4. Save the file. The changes will take effect the next time you log in, i.e. you need to log out and log in again.

D.2 Running the MECA simulation

Make sure you have installed the KAoS/Brahms bridge as described in section D.1. To start the simulation and the ePartner GUI, first start KAoS as described in section D.2.1. When KAoS is running, execute the steps in section D.2.2 to start the ePartner GUI.

D.2.1 Starting KAoS with the MECA ontologies

1. Execute the `mecasimulation/run-kaos.bat` file, or use the `.sh` file when running on Mac OS X.
2. Leave this command window open to keep KAoS running.

D.2.2 Running the ePartner GUI

1. Make sure KAoS is running.
2. Execute the `mecasimulation/run-with-gui.bat` file, or use the `.sh` file when running on Mac OS X.

D.2.3 Running in the Brahms Composer

1. Make sure KAoS is running.
2. Copy `mecasimulation/lib/mecasimulation.jar` to the Brahms `deploy` folder.
3. Start the Brahms Composer.
4. Import `mecasimulation/src_brahms/nl/tno/hf/meca/main.b`.
5. Open the menu `Build > Build Options`.
6. Add the folder `kaosbrahmsbridge/Brahms Policy Library` both as a source folder and as a library.

D.3 Using the KAoS/Brahms bridge

The KAoS/Brahms bridge consists of three main components: the Brahms Policy Library, the KAoS Gateway Agent and the Ontology Builder.

The Brahms Policy Library provides a generic API Brahms agents can use to query policy services. This library is backed by the KAoS Gateway Agent, which provides the implementation necessary to communicate with KAoS. However, to define policies about concepts in the Brahms simulation, KAoS needs an ontology describing what those concepts are. This ontology can be automatically generated from the Brahms model using the Ontology Builder.

A guide to using the bridge in a new or existing Brahms model is presented in this section.

D.3.1 Using the bridge in a Brahms model

1. Make sure you have installed the KAoS/Brahms bridge as described in section D.1.
2. Open a (new or existing) Brahms model.
3. Open the menu `Build > Build Options`.
4. Add the folder `kaosbrahmsbridge/Brahms Policy Library` both as a source folder and as a library.
5. Use the Ontology Builder to construct an ontology that corresponds to your Brahms model. See also to section D.3.2.
6. Place the generated ontology files at the URL you specified in the Ontology Builder properties.
7. Modify the `ontology.url.classes` property in the `kaosgateway.properties` file, located in you Brahms configuration folder. The variable should point to the URL of the ontology containing the Brahms classes.
8. Start KAoS and open the configuration file generated by the Ontology Builder.

D.3.2 Using the Ontology Builder

1. Adjust the settings in the properties file `ontologybuilder.properties` to your needs.
2. Using Ant to run the `runxslt` target in the `kaosbrahmsbridge/Ontology Builder/build.xml` build file to construct the ontology files.
3. Run the `buildkaosconfiguration` target using Ant to construct a KAoS configuration file that contains all agent instances of your model and imports the ontology files.

Bibliography

[1] J. Bradshaw, P. Beautement, M. Breedy, L. Bunch, S. Drakunov, P. Feltovich, R. Hoffman, R. Jeffers, M. Johnson, S. Kulkarni, et al. Making agents acceptable to people. *Intelligent Technologies for Information Analysis: Advances in Agents, Data Mining, and Statistical Learning*, pages 355–400, 2004.

[2] Brahms Authors. Brahms Website. `http://www.agentisolutions.com/`, 2009.

[3] L. Breebaart, A. Bos, T. Gran, A. O. Soler, M. Neerincx, N. Smets, J. Lindenberg, U. Brauer, and M. Wolff. The MECA Project: Using An OWL/RDF Knowledge Base To Ensure Data Portability For Human Space Mission Operations. `http://www.crewassistant.com/papers.asp?InfoId=71`, 2008.

[4] N. Damianou, N. Dulay, E. C. Lupu, and M. S. Sloman. *Ponder: A Language for Specifying Security and Management Policies for Distributed Systems Version 2.3.* Imperial College of Science, Technology and Medicine, Department of Computing, 2002.

[5] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. `http://www.w3.org/TR/owl-features/`, 2004.

[6] Foundation for Intelligent Physical Agents. *FIPA Communicative Act Library Specification*, 2002.

[7] Institute for Human and Machine Cognition. KAoS: User Guide, 2008.

[8] M.A. Neerincx. ePartners for accommodating cognitive and affective load of planetary explorers. `http://www.crewassistant.com/docs/pub/TPS_Workshop/ePartnerforLoadandEmotionAccomodation_final.pdf`, 2007.

[9] MECA Consortium. MECA Website. `http://www.crewassistant.com/`, 2009.

[10] M. Neerincx, J. Lindenberg, N. Smets, T. Grant, A. Bos, A. Olmedo Soler, U. Brauer, and M. Wolff. Cognitive engineering for long duration missions: human-machine collaboration on the Moon and Mars. In *Proceedings of 2nd IEEE International Conference on Space Mission Challenges for Information Technology*, pages 40–46, 2006.

[11] M. Sierhuis, W. J. Clancey, and R. J. J. van Hoof. *Multi-Agent Programming (to appear)*, chapter Brahms: An Agent-Oriented Language for Work Practice Simulation and Multi-Agent Systems Development. Springer, 2009.

[12] A. M. Tiryaki, S. Öztuna, O. Dikenelli, and R. C. Erdur. *SUNIT: A Unit Testing Framework for Test Driven Development of Multi-Agent Systems*, pages 156–173. Springer Berlin/Heidelberg, 2007.

[13] A. Uszok, J. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken. KAoS policy management for semantic web services. *IEEE Intelligent Systems*, 19(4):32–41, 2004.

[14] J. van Diggelen, J. Bradshaw, T. Grant, M. Johnson, and M. Neerincx. Policy-Based Design of Human-Machine Collaboration in Manned Space Missions. In *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2009.

[15] R. J. J. van Hoof. Brahms Java API. `http://www.agentisolutions.com/documentation/vmapi`, 2009.

[16] Wikipedia Editors. Ontology (information science). `http://en.wikipedia.org/wiki/Ontology_(computer_science)`, 2009.

[17] M. Wooldridge. *Introduction to multiagent systems*. John Wiley & Sons, Inc. New York, NY, USA, 2001.