FAKULTÄT FÜR !NFORMATIK

RASCALLI Platform: A Dynamic Modular Runtime Environment for Agent Modeling

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Christian Eis
Matrikelnummer 9325145

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer/Betreuerin: Ao. Univ.-Prof. Dipl.-Ing. Dr. Harald Trost
Mitwirkung: Mag. Dr. Brigitte Krenn

Wien, 22.10.2008 _____ _____
(Unterschrift Verfasser) (Unterschrift Betreuer)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Acknowledgements

I greatly value the support I received from a number of people and organizations over the last few years and, in some cases, my whole life. However, I also value conciseness, so I will keep this section short.

Many thanks go to my parents for their seemingly unlimited trust and patience, my wife for being very patient during these last years and for attempting to push me a bit from time to time, and to Markus Mayer for pestering me with questions about my progress at university, although he seems to have gotten tired of it in the last couple of years.

I want to thank my advisors, Harald Trost and Brigitte Krenn for the obvious reasons, and my colleagues at the RASCALLI project partners (OFAI[1], SAT[2], DFKI[3], Radon Labs[4], NBU[5] and Ontotext[6]) for implementing various parts of the RASCALLI system[7], so that I had something to integrate.

Many thanks go to my employers, Research Studios Austria Forschungsgesellschaft[8] and in particular the Smart Agent Technologies Studio, as well as the Austrian Research Centers GmbH[9], and to their sponsors, the Federal Ministry of Economics and Labour and the Federal Ministry of Science and Research of the Republic of Austria.

This work is part of the RASCALLI project[10], which is funded by the European Commission under the Sixth Framework Programme[11].

[1] Austrian Research Institute for Artificial Intelligence, `http://www.ofai.at`

[2] Smart Agent Technologies Studio – Research Studios Austria, `http://sat.researchstudio.at`

[3] German Research Center for Artificial Intelligence, `http://www.dfki.de`

[4] http://www.radonlabs.de

[5] New Bulgarian University, `http://www.nbu.bg`

[6] Ontotext Lab, Sirma Group, `http://www.ontotext.com`

[7] The individual contributions are indicated as they appear in the document.

[8] `http://www.researchstudio.at`

[9] `http://www.arcs.ac.at`

[10] European Commission Cognitive Systems Project FP6-IST-027596-2004 RASCALLI. `http://www.ofai.at/rascalli`

[11] `http://www.cognitivesystems.eu`

Kurzfassung

Wir beschreiben die Architektur und Implementierung der RASCALLI Plattform, einer Laufzeit- und Entwicklungsumgebung für Softwareagenten. Der zentrale Beitrag dieser Arbeit ist die Anwendung moderner komponentenbasierter Entwicklungsmethoden auf die Implementierung von Agenten, die es erlaubt, solche Agenten aus einer Menge wieder verwendbarer Komponenten zusammenzufügen. Mehrere unterschiedliche Agenten können in einer einzelnen Instanz der Plattform gleichzeitig ausgeführt werden, wodurch die Evaluierung und der Vergleich von einzelnen Komponenten, sowie von kompletten Agentenarchitekturen ermöglicht wird. Schließlich erleichtert die Service-orientierte Architektur der Plattform die Integration von externen Komponenten.

Abstract

We introduce the RASCALLI platform, a runtime and development environment for software agents. The major contribution of this work is the application of modern component-based software engineering techniques to agent development, enabling the construction of agents from a set of reusable components. Agents of different kinds can be implemented and executed within a single runtime environment, allowing for effective evaluation and comparison of individual agent components as well as entire agent architectures. Finally, the platform's service-oriented architecture greatly facilitates the integration of external and legacy components.

Contents

Chapter 1

Introduction

Agent-based software engineering is becoming an ever more prominent branch of software development, not only in the artificial intelligence and cognitive science communities, but also in many industrial and commercial settings. There is, however, evidence that more recent advancements in software development have not yet found their way into agent engineering environments and processes. Examples include object-oriented design methodologies and iterative development ([Bry03]) and aspect-oriented programming ([GL08]). To quote from the introductory text of the 9th International Workshop on Agent Oriented Software Engineering (AOSE, 12-13 May, 2008, [AOS08]):

> Since the mid 1980s, software agents and multi-agent systems have grown into a very active area of research and also commercial development activity. One of the limiting factors in industry take up of agent technology is however the lack of adequate software engineering support, and knowledge in this area.

Component-based software development is another technology that has not yet been fully adopted in agent-based systems (with the possible exception of multi-agent systems). Even though many agent environments and cognitive architectures make use of modules to structure agents (e.g. into behavior modules), the full power of software components, including re-use ([dSdM08]) and dynamic composability, is seldom used.

This document introduces the RASCALLI platform for component-based agent development. It builds on state of the art technology (OSGi) and a modular system architecture to implement a fully dynamic environment for agents that are composed from a set of building blocks.

1.1 Component-based Development

The utility of decomposition and modularity is well-understood in the computer sciences. Virtually all programming languages support the concept of reusable software modules or libraries, and object-oriented programming goes even a bit further by encapsulating data and functionality in well-separated entities (classes). The field of component-based software engineering (CBSE, see [SH04], [PS96], [Cle96]) has evolved beyond object-oriented development to provide a more general notion of re-usable software components. To put it simple, a software component is an independent collection of code with a well-defined interface and contract for collaboration with other components. These components can be used as units of software development, but also as deployment units, in the sense that a running system is composed of a set of components.

Even though the idea of component-based software has been around for quite a while ([McI68]), most modern programming languages do not (yet) explicitly support the notion of software components. For example, Java has the notion of packages, which might be defined as software components from a development point of view, but not from a runtime perspective. An effort to correct this situation is currently being undertaken in the form of Java Specification Request 277 [JSR].

Several variants of component-based middleware technology exist, including CORBA[1], (D)COM[2], various implementations of Remote Procedure Call (RPC), such as RMI[3], and many more. The most recent and currently most popular incarnation is Service Oriented Architecture (SOA)[4] and is usually based on Web Services[5]. All of these technologies build on the notion that an application is composed of distributed service components. However, these kinds of component-based applications have several disadvantages:

- Interfaces between components are defined using a special interface description language. Special tools are then used to implement this interface in the programming language of choice.

- Each component is a separate application, often deployed to different host environments. Running and maintaining such a distributed system

[1] Common Object Request Broker Architecture, `http://www.corba.org/`

[2] (Distributed) Component Object Model, `http://www.microsoft.com/com`

[3] Remote Method Invocation, `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`

[4] There does not seem to be a definite work on SOA, so we refer to Wikipedia: `http://en.wikipedia.org/wiki/Service-oriented_architecture`

[5] `http://www.w3.org/2002/ws/`

is naturally difficult.

- Network communication imposes a performance overhead on component interaction.
- Since each component is a separate application, the granularity of the components is quite large.

In contrast to the aforementioned technologies, OSGi provides a thin layer on top of the Java Virtual Machine (JVM), that extends Java with functionality for dynamic[6] component-based and service-based applications. This approach mitigates the disadvantages listed above:

- Components interact via simple Java interfaces. No special languages or tools are required.
- All components are part of a single executable application (but components can be added and removed at runtime). However, it is quite easy to integrate external components, using one of the mentioned middleware technologies, and making it available as a service to other components within the OSGi application.
- There is no communication overhead, since Java objects interact directly within a single JVM.
- The granularity of the components can be arbitrarily chosen depending on the implementation requirements.

1.2 Component-based Agent Development

Given the success of modularization and software reuse in general software engineering, it is not very surprising that these concepts have also been employed for agent development. Many agent frameworks and cognitive architectures make use of modules in one way or another, including prominent examples such as the Subsumption Architecture [Bro86] , ACT-R [And93] and AKIRA (see section 3.2 on page 23). However, they are not dynamic component-based systems in the sense described in the previous section.

Multi-agent systems (MAS, see section 3.1 on page 21) fill this gap in so far as a complete application is built from a selection of interacting agents. From a software engineering point of view, a MAS is a component-based

[6] *Dynamic* in this context means that components can be added and removed at runtime, without restarting the entire application.

system, with each individual agent being a component. However, MAS have the same drawbacks as the component-based middleware systems outlined above.

1.3 RASCALLI Platform

The RASCALLI platform is a runtime and development environment for agent development. Building on OSGi, it provides a dynamic component-based architecture, within a single application.

In its current state, the platform is not meant for industrial use, but rather as a tool for cognitive agent research. The goal is to allow a team of researchers to implement and experiment with different kinds of agents in a single environment and to allow these agents to be assembled from a pool of components. New agent components can be added to and removed from a running platform instance, allowing for a dynamic development process.

The platform also tries to minimize the effort of integrating external or legacy components. Such components are often used in a research context, where a given project builds on the outcome of previous projects and the resources for re-implementing everything from scratch are not available.

While the RASCALLI platform supports the execution of multiple agents, as well as the communication between those agents, it is not a multi-agent system. A MAS is typically a single application that makes use of a distributed and dynamic agent model to solve a given problem. However, it would be possible and perhaps even quite promising to integrate existing multi-agent systems (e.g. JADE, section 3.1.1.1 on page 22) or parts of such systems with the RASCALLI platform.

It is important to note that the RASCALLI platform is not bound to a specific cognitive architecture. Instead, the platform explicitly supports the idea of implementing (or integrating) a number of different cognitive or non-cognitive agent architectures so that they can be compared and evaluated or used for different purposes, when appropriate.

The RASCALLI platform is currently a proof-of-concept implementation of the basic ideas described in this document. Promising directions for future work include (for more details see chapter 9):

- the extension of the agent development environment,
- the integration of existing agent development frameworks such as BOD (see sections 3.3 and 9.2),
- the integration of existing multi-agent systems or the implementation of FIPA-compliant components, and

- the integration or re-implementation of existing cognitive architectures.

1.4 Motivation

The RASCALLI platform has been developed as the central software system of the RASCALLI project, based on the objectives and requirements set forth in chapter 2. In short, we aimed at an environment that would allow us to implement and experiment with cognitive agents assembled from simple components. It should be possible to re-use components for multiple different agents and also to add components to an agent at runtime.

We also struggled with a very heterogeneous aggregation of existing and newly built system components and tried to minimize the effort of integrating and managing the entire system. The first prototype of a software platform for this system proved very hard to handle in this respect.

After extensive research into available systems and technologies, and with Java being the programming language of choice for the RASCALLI project, we finally arrived at the idea of implementing a dynamic, component-based agent environment on top of OSGi. One of the main goals for the platform and for the agents developed within the platform, was extensibility both in the sense of being able to add newly implemented components and in the sense of being able to easily integrate existing components. After all, implementing an entire agent system or cognitive architecture from scratch would be a quite ambitious undertaking, especially with so many systems already being available.

Finally, we wanted to demonstrate that the application of modern software engineering techniques to cognitive agent research is valuable and feasible. Accordingly, the resulting software platform will be made available to the general public so that other projects can build on and extend it.

1.5 Document Structure

This document is structured as follows:

Chapter 2 outlines the RASCALLI project objectives and the requirements for the software platform derived from those objectives, as well as some additional constraints.

Chapter 3 gives an overview of related work in the areas of agent systems, cognitive architectures and agent development methodologies, discussing how they relate to the RASCALLI platform and which requirements are not met by these systems.

Chapter 4 introduces the RASCALLI platform features and software architecture while chapters 5 through 7 provide more detail about the three platform layers.

Chapter 8 presents an overview of the entire RASCALLI system, from the user's point of view.

Finally, chapter 9 summarizes the document and provides an outlook into promising future directions.

Chapter 2

Project Objectives and Requirements

This chapter starts by defining a set of terms to be used throughout the document. It continues with a short introduction of the RASCALLI project, describing its primary goals and objectives, insofar as they are relevant for the design and implementation of the RASCALLI platform. We then list some additional constraints found during the first project year, which also had an influence on the final platform architecture. Finally, we derive a list of requirements for the platform implementation from these goals and constraints.

2.1 Terms and Definitions

Cognition: There is little agreement in the cognitive science community as to the exact definition of *cognition*. However, most researchers in this area would agree that cognitive systems (natural and artificial) have at least some of the following properties: perception, action, memory, learning and sometimes planning. For the purpose of this thesis, we abstain from suggesting our own definition and instead concentrate on the just mentioned set of properties, calling them *aspects of cognition*. A cognitive system is thus a system that comprises one or more of these aspects.

Agent: As with the concept of *cognition*, the term *agent* has no clear and generally accepted definition. Many authors assign a number of attributes to an agent, including persistence, autonomy, social ability, reactivity, proactivity, ability to adapt, ability to learn and sometimes mobility. From the RASCALLI platform point of view, any piece of

software could be an agent, as long as its implementation conforms to the technical framework outlined in this document. However, the term *agent* is used to refer to RASCALLI agents, as defined below.

Multi-agent system: A multi-agent system (MAS) is a system composed of multiple agents which interact in order to solve some global goal. MAS are usually distributed systems and each individual agent would not be capable of fulfilling its tasks without the help of other agents.

Complete agent: "A complete agent is an agent that can function naturally on its own, rather than being a dependent part of a Multi-Agent System (MAS)." [Bry03, page 1]

Complex agent: "A complex agent is one that has multiple, conflicting goals, and multiple, mutually-exclusive means of achieving those goals. Examples of complete, complex agents are autonomous robots, virtual reality (VR) characters, personified intelligent tutors or psychologically plausible artificial life (ALife)." [Bry03, page 1]

Embodied agent: We think of an embodied agent as an agent that lives in a defined environment (e.g. the physical world) and interacts with this environment via sensors and effectors. The sensors and effectors are part of the agent's body and function as an interface between the agent and the environment. An example of embodied agents are autonomous robots.

Virtually embodied agent: A virtually embodied agent is one that lives in a virtual world (as opposed to the physical world). A virtual world could be a simulation of a physical environment, such as a game engine, or a real world virtual environment, such as the Internet, which is used by real world agents, such as humans. An agent living inside a virtual environment has, of course, a virtual body instead of a physical one, which consists of a set of software modules that act as sensors and effectors in the virtual world.

We argue that most of the arguments for embodied cognition apply equally well to virtual embodiment. The rational is that, in contrast to classical purely symbolic AI-systems, the agent has a virtual equivalent of a physical body, which is an integral part of the agent and constitutes some of its cognitive properties.

RASCALLI agent: A RASCALLI agent is a virtually embodied, complete, complex agent augmented with aspects of cognition. Throughout

the remainder of this document, the term *agent* refers to RASCALLI agents, unless specified otherwise.

Rascalli, Rascalla, Rascallo: We use the term *Rascalli* to refer to multiple RASCALLI agents, and *Rascallo* or *Rascalla* to refer to a single male or female RASCALLI agent, respectively[1].

2.2 The RASCALLI Project

The RASCALLI project[2] is a joint project of six European research and industry partners, combining expertise in cognitive architectures, natural language processing, multi-modal generation, machine learning, information extraction, semantic web technology, general web technologies and 3D graphical user interfaces. RASCALLI stands for Responsive Artificial Situated Cognitive Agents Living and Learning on the Internet. Annex I of the project contract states that

> The overall goal of the RASCALLI project is to develop an artificial cognitive system that allows human and computer skills to be combined in such a way that both abilities can be optimally employed for the benefit of the human user. We develop and implement a system platform and toolkit based on which responsive artificial situated cognitive agents, the Rascalli, can be created and trained by the human user in order to perform certain tasks. Their cognitive system, knowledge and skills enable them to acquire and constantly improve their knowledge through the Web or through communication with people and other Rascalli. To achieve this goal, Rascalli have a built-in cognitive model and base knowledge about people, themes, Internet, time and communication. They know about general preferences of humans, about document structures, encodings, languages, pictures, metadata, and gradually learn about search engines, archives, and web classification systems. They also have built-in skills: They are specialists in WWW navigation, they can communicate with each other and the user via email, chat and animated conversation, and they can play games. Moreover Rascalli have an enormous

[1] Note that the gender of an agent only determines the looks of the virtual character representing the agent in the 3D client user interface and does not influence any other aspect of the agent.

[2] European Commission Cognitive Systems Project FP6-IST-027596-2004 RASCALLI. `http://www.ofai.at/rascalli`

> memory capacity, however, their memory is cognitively structured for fast associative access. Rascalli have the ability to distinguish Rascalli and users they have already met in the past from novel encounters, and accordingly can adapt their behaviour and communication strategies, i.e. seek the company of those they have had positive experiences with or avoid disagreeable encounters, with (dis)agreeability being determined by means of appraisals. [RAS05, page 4]

One of the primary goals of the project is to integrate a cognitive architecture (DUAL/AMBR, [Kok94], [KPss]) with a set of software tools for accessing and manipulating services on the Internet, such as search engines, knowledge bases and communication services. These tools act as sensors and effectors in the virtual environment of the Internet. Each tool is a separate component and agents can be equipped with a subset of existing tools in order to acquire different skill sets. Ideally, it would be possible to add or remove tools at runtime in order to change an agent's abilities. In addition, the users can train their agents so that these will become experts in a specific sub-domain of the chosen application domain (which is popular music information retrieval).

Another objective is to be able to experiment with different implementations of cognitive aspects, such as action selection, memory and learning. Again, the modular nature of the system should allow for the selection and integration of certain cognitive functions with the available tools.

RASCALLI agents have a presence on the Internet, which means that they must be active and available all the time and therefore should run in a server environment instead of a desktop application that is only active when the user's computer is running. For example, a Rascallo should be able to monitor the Internet (e.g. RSS feeds chosen by the user) for interesting new information and make that information available to the user. What's more, the Rascallo should notify the user of newly available information that is potentially interesting.

Rascalli interact with their users via a set of communication channels, including a client-based 3D virtual character interface with speech and gesture output and an instant messaging integration (Jabber). They are also able to communicate with other Rascalli existing in the same environment. One of the project goals is to build a community of agents which have been trained by their users to be experts in different knowledge domains and to allow them to help each other in fulfilling their information retrieval tasks.

In order to allow the Rascalli to actually live on the Internet, form a community and proactively assist their users, we had to implement a server

platform that allows multiple users to instantiate multiple Rascalli, which then run independently inside the platform.

2.3 Additional constraints

For the realization of these objectives, system integration turned out to be a major roadblock, due to the following reasons:

- Even though Java was chosen as the main implementation language for the project, some project partners have no or little experience with Java development.
- In order to avoid re-implementation, we had to integrate existing components from previous projects. These components are based on a wide range of technologies, such as different programming languages (e.g. Perl, Java, Lisp) and even native binaries for different operating systems (Linux and Windows).
- Initial attempts at providing a development environment that integrates all of these components and can be replicated on each developer's machine proved to be difficult to use and keep up-to-date.

Figure 2.1 on the next page shows the external components that need to be integrated with the platform for one of several kinds of agents currently implemented[3]. It shows only those components that are directly connected to the platform, but omits other components, such as databases or Internet services, which are used by the shown external components or by components implemented within the platform (e.g. Wikipedia is not displayed, even though a component exists within the platform that accesses it). The figure also shows, for each component, the kind of technology used for the component's implementation and/or integration. This should give a rough impression of the integration effort required by such a setup.

2.4 Platform Requirements

Based on the project objectives and constraints outlined above, we arrived at the following **set of requirements** for our software platform:

- Support the execution of various agents, belonging to different users,

[3]Namely the Simple Music Companion, described in section 7.2 on page 60.

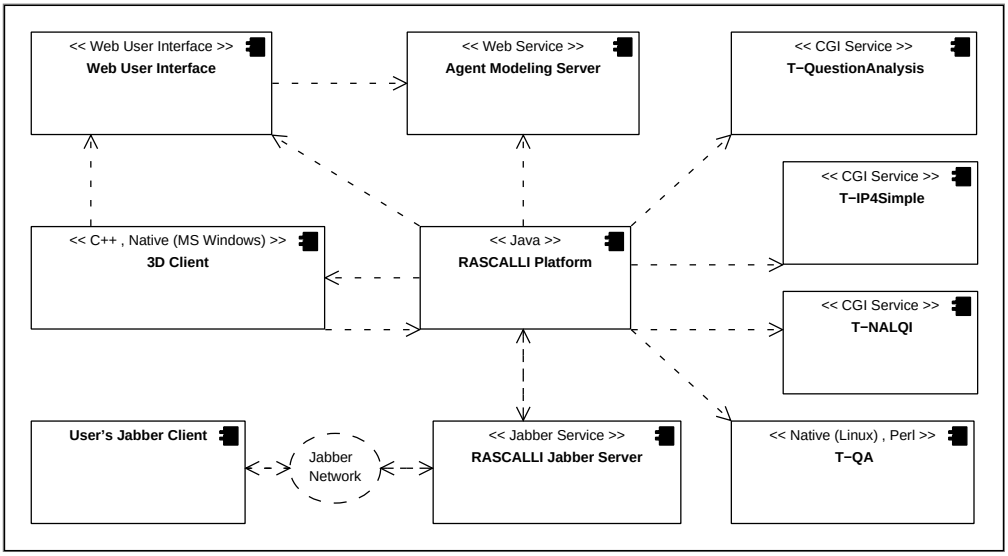

Figure 2.1: Overview of the external RASCALLI components used by the Simple Music Companion, including implementation technologies. Single and double arrows denote uni- and bi-directional communication between components, respectively.

- support agent-to-agent and agent-to-user communication,
- allow developers to implement diverse agents based on shared components (this also means that multiple versions of each component can exist at the same time),
- integrate external and legacy components with minimal effort,
- build agents in a modular, component-based fashion,
- build the platform itself in a component-based, extensible fashion, and
- build a system that can be extended and improved dynamically at runtime.

2.5 Summary

The RASCALLI project aims at the implementation of virtual agents that perform tasks related to accessing and processing of information from the Internet and domain-specific knowledge bases. The agents are constructed from

a set of cognitive functions and a set of sensor and effector tools to access Internet services in an autonomous and proactive manner, taking into account the interests and preferences of the individual user. Many of the functions and components are based on existing software from previous projects that had to be integrated with the RASCALLI software system.

Based on the project objectives and system integration constraints, we have defined a set of requirements for the underlying software environment, the RASCALLI platform.

Chapter 3

Related Work

Of the large number of existing frameworks and technologies for agent development, this chapter will present those that had the most influence on the design of the RASCALLI platform or that seem to offer the most promising directions for future work. We also present some frameworks in order to draw a clear line between existing wide-spread technologies and the RASCALLI platform.

3.1 Multi-Agent Systems

Multi-agent systems (MAS) are typically used in complex domains, where difficult or very large problems are broken down into smaller subproblems in order to make those problems solvable. A possibly large number of simple agents interact in order to fulfill the entire system's goals. According to [Syc98, page 80],

> The characteristics of MASs are that (1) each agent has incomplete information or capabilities for solving the problem and, thus, has a limited viewpoint; (2) there is no system global control; (3) data are decentralized; and (4) computation is asynchronous.

Since MAS perform distributed computation, communication and coordination among agents are critical for the success of the entire system. Today, multi-agent systems are used in a wide variety of application domains, including simulation, communication, process control and financial systems.

3.1.1 FIPA

"FIPA is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies."[1] The specified standards deal mostly with agent communication (e.g. Agent Communication Language, ACL) and agent management.

3.1.1.1 JADE

The Java Agent DEvelopment Framework (JADE, [BPR01])[2] is the most prominent example of a FIPA-compliant agent development framework. It is a Java-based middle-ware for distributed multi-agent systems and provides basic services and infrastructure, such as

- agent life-cycle management,
- agent mobility,
- agent communication (using ACL),
- white and yellow page services, and
- security.

Furthermore, it implements a number of graphical tools for debugging and monitoring of running agent systems. Due to its small footprint, JADE can also be used on mobile devices, making it an ideal solution for highly dynamic applications in a mobile context.

3.1.2 Discussion

While the RASCALLI system could have been implemented as a multi-agent system, the scope of such systems is quite different from the one targeted by RASCALLI. In particular, RASCALLI has no requirement for (physical) distribution of the services. Also, in RASCALLI we aimed at a different granularity regarding the modularization of the software. While MAS are typically used to split a single complex application into more manageable parts, we were looking for modularity on the agent level.

Even though, the RASCALLI project also has the goal of implementing an agent community. On this level, some of the FIPA standards (e.g. ACL) might be very useful, even though much simpler approaches are sufficient

[1] `http://www.fipa.org/`, retrieved October 6, 2008
[2] `http://jade.tilab.com/`

for the scope of the RASCALLI project itself. Thus, one direction for the future development of the RASCALLI platform is the integration of (parts of) JADE with the platform.

3.2 AKIRA

"AKIRA is an open source framework designed for parallel, asynchronous and distributed computation, on the basis of some general architectural principles which are inspired by modular organization in biological systems." [PC07, page 1]

One of the most interesting aspects of AKIRA is the use of a limited amount of energy shared between the individual modules of an agent (daemons). The daemons can exchange this energy, leading to very interesting dynamical effects. Agent communication is organized via a blackboard infrastructure.

3.2.1 Discussion

The use of AKIRA for the RASCALLI project was hindered by various factors:

- AKIRA is based on a very specific, biologically inspired architecture that is not necessarily a good match for some of the components that were to be integrated in RASCALLI (e.g. DUAL/AMBR).
- AKIRA aims at the development of single modular agents, but not at the development of agent communities.
- AKIRA is implemented in C++ and the language of choice for RASCALLI was Java.

There are, however, some aspects of AKIRA that would be interesting to re-implement on top of the RASCALLI platform, most importantly, the concept of a shared and limited energy pool. This approach could add the following benefits to the RASCALLI platform:

Limited resource consumption: For a community of agents living on the Internet, access to some resources must be limited (beyond the physical limitations of processing hardware and network connection). For example, free access to Internet services is often limited, while payment options offer large-scale access.

Additional constraints on action selection: Agents would be required to more carefully select their actions, based on energy cost and expected benefit.

Interesting multi-agent dynamics: Limited resources might lead to interesting effects in a society of expert agents. For example, an agent might specialize on performing some expensive action, such as crawling and analyzing a large set of web pages at regular intervals, and provide access to the extracted information as a service to other agents.

3.3 Behavior-Oriented Design

Behavior-Oriented Design (BOD, [Bry01])[3] is a development methodology and software architecture for modular software agents. Its goal is to apply the advantages of object-oriented software engineering to the design and implementation of complex, complete agents. Like the RASCALLI platform, it does not aim at the development of multi-agent systems.

The core of BOD is a software development process for the implementation of software agents. In particular, it supports the developer in finding the right granularity of modularization by applying methodologies from object-oriented design to agent development:

> The advantages of treating a software system as modular rather than monolithic are well understood. Modularity allows for the problem to be decomposed into simpler components which are easier to build, maintain and understand. In particular, the object-oriented approach to software engineering has shown that bundling behavior with the state it depends on simplifies both development and maintenance. [Bry03, page 1]

3.3.1 BOD design process

BOD splits an agent into behaviors and (reactive) plans. Each behavior is implemented as an object, containing methods for sensory and action primitives. A behavior also encapsulates any state necessary to perform these primitives.

At the core of BOD is an action selection mechanism based on hierarchical plan structures (POSH action selection[4]). These plans comprise:

[3] `http://www.cs.bath.ac.uk/ai/AmonI-sw.html`

[4] `http://www.cs.bath.ac.uk/~jjb/web/posh.html`

POSH Primitives: An agent's primitive actions and senses, which are implemented as methods of a behavior class.

POSH Aggregates: These build the hierarchy of the plan:

- The root of a POSH plan is called a *drive-collection* and determines the agent's top-level goals. It is, in effect, a competence with some additional properties (e.g. it never terminates).
- *Competences* are prioritized lists of actions, which can contain primitives and action patterns.
- *Action patterns* are simple sequences of primitives and other action patterns.

BOD starts with an initial decomposition step, during which first lists of action sequences, sensory and action primitives, and high-level drives are compiled. The action sequences and drives form the initial reactive plan structures. The decomposition into behavior modules is determined based on the state required by the sensory and action primitives. The goal is to combine those primitives into a behavior that share some common state. This is analogous to object-orient design, where system functions are separated into classes in a similar fashion.

Following this first step, the agent is implemented in an iterative process, similar to modern agile software development processes. During each iteration, part of the specification derived from the initial decomposition is implemented and tested. Then the specification is revised, following a set of rules provided by the BOD development process.

3.3.2 Discussion

Similar to the RASCALLI platform, Behavior-Oriented Design aims at a modular development of software agents. Behaviors can be reused for different agents, running in a single runtime environment, where each agent has its own reactive plan.

However, BOD does not address the communication requirements of the RASCALLI project (agent-to-agent and agent-to-user communication). Furthermore, the integration of external components is not explicitly addressed (except insofar as the program code to integrate a particular component can be re-used by multiple developers). Finally, BOD does not address the issue of a dynamically extensible software system.

For the RASCALLI platform, we want to take the concept of component-based, modular development one step further, beyond re-use at the code level.

Components should be separate entities that can be worked on independently, and added to and removed from a running system.

The BOD development process can be easily applied to agent development in the RASCALLI platform. Also, the RASCALLI platform specifically allows for the implementation of multiple agent architectures instead of proposing or focusing on a single one. In fact, it might be very interesting, to build an agent architecture based on BOD within the platform. A possible implementation scenario is described in section 9.2 on page 75.

3.4 Pogamut

Pogamut[5] is a development environment for agent development. It specifically targets the Unreal Tournament[6] game engine, using it as an environment for virtual agents, complete with physics and graphical representation. Its most interesting aspects, from the RASCALLI point of view, are the fact that it provides an integration with the Netbeans Java IDE, complete with online modifications and debugging of agents in the target environment; and that they have integrated POSH action selection as the basis of one of their types of agents (see section 9.2 on page 75 for a similar integration scenario for the RASCALLI platform).

3.5 Summary

After defining the requirements for the RASCALLI software system, we investigated a number of agent environments and development methodologies. While none of these fulfilled all of our requirements, certain aspects of these systems gave valuable input for the design of the RASCALLI platform. Even more important, they provide inspiration and software modules for possible future extensions of the platform.

[5] `https://artemis.ms.mff.cuni.cz/pogamut/tiki-index.php?page=HomePage`
[6] `http://planetunreal.gamespy.com/`

Chapter 4

RASCALLI Platform

This chapter gives a high-level overview of the platform's software architecture. We first describe the platform features and how they relate to the project objectives and requirements set forth in chapter 2. This is followed by a section about the software architecture itself and the technologies chosen to implement the architecture. Alternative approaches are explored where appropriate. In the final section, we explore the relation of the RASCALLI platform to other agent-based systems, by mapping the platform to the Agent-Based Systems Reference Model.

4.1 Platform Features

The RASCALLI platform has a number of features, which have been derived from the RASCALLI project objectives and the requirements for the platform, as detailed above. Table 4.1 on the next page is an overview of the features, which are described in more detail in the rest of this section.

4.1.1 Multi-Agent

One of the goals of the RASCALLI project is to have a number of agents with different abilities and also different experiences, based on the interaction with their users. We also want to provide the basis for implementing a community of agents that help each other fulfill their tasks. Therefore, the RASCALLI platform supports the implementation and execution of multiple agents.

4.1.2 Multi-Agent-Architecture

In order to be able to experiment with different kinds of agents, which might be rather similar or very dissimilar, the platform allows for the implemen-

tation of arbitrary agent architectures. Furthermore, a component-based approach to agent development is adopted (see below), which allows for more fine-grained differentiation of agents within each agent architecture. All agents are able to interact with each other.

Feature	**Description**
Multi-agent	Multiple agents can be executed simultaneously within a single platform instance.
Multi-agent-architecture	Different agent architectures can be implemented within the platform and agents of these different architectures can interact with each other.
Multi-user	Each agent has a single user – each user owns one or more agents.
Shared platform	A single platform environment is shared by multiple agents, users and agent developers.
Communication	Agents can communicate with their users and other agents.
Component-based architecture	Agents are assembled from reusable components. New agent components and agents can be deployed at runtime.
Extensibility	The platform itself is modular and can be extended, even at runtime.
Multi-version	Different versions of software components can exist at the same time within a single running platform instance.

Table 4.1: RASCALLI Platform Features

4.1.3 Multi-User

RASCALLI agents are created and owned by human users. Consequently, if the platform supports a community of agents, it must also support a community of users.

4.1.4 Shared Platform

A single instance of the RASCALLI platform is shared by multiple users and agents. User accounts and agents can be added to or removed from the platform at runtime. The platform is also designed to be shared by a team of agent developers.

It should be noted that this feature does not necessarily follow from the multi-agent and multi-user properties. For example, each user might create its agent (or agents) locally and they might then communicate with each other over the Internet in a peer-to-peer fashion.

However, having a single shared environment greatly facilitates the following activities:

System integration: External (and possibly legacy) components need to be integrated only once and are then available to all agents running in the platform in an easy-to-use manner. There is no need to duplicate the entire software environment on multiple developer machines. This could, of course, also be achieved with a distributed service-oriented architecture, where each legacy component is hidden behind a service. However such a system is much harder to implement and maintain than a less-distributed system. For example, bi-directional communication between components is far easier to implement in a shared environment than in a distributed system.

Agent communication: A single shared environment offers more possibilities for agent communication than a distributed one.

Furthermore, the agents are supposed to "live on the Internet" and perform certain tasks even when their users are offline. Consequently, the users' desktop computers are not an ideal habitat for the agents.

4.1.5 Communication

The project objectives explicitly state that RASCALLI agents communicate with their users and other agents in order to fulfill their tasks. Implementing a selection of communication channels within the platform allows various agents to share these communication components.

Agent-to-agent communication can be implemented on the Java level, since all agents run within a single runtime environment. Alternatively, agents can communicate via instant messaging (this opens the possibility to use multiple, distributed platform instances in the future).

Several channels for agent-to-user communication have been implemented (currently a proprietary protocol for the 3D client interface, Jabber instant messaging and web-based communication).

4.1.6 Component-Based Architecture

In addition to the general advantages of component-based development, a modular architecture explicitly supports the goal to experiment with different kinds of agents. While the platform cannot enforce a component-based approach (an agent could be implemented as a single large Java class), its architecture certainly encourages and facilitates the use of components. This results in agents that can be assembled from these components in a Lego-like manner.

4.1.7 Extensibility

The platform itself is built in a component-based fashion and can therefore be easily extended, even at runtime.

4.1.8 Multi-Version

In order to support the concurrent development of different kinds of agents based on shared components in a single environment shared by multiple developers, the platform supports the execution of multiple versions of the same components at the same time.

4.2 Software Architecture

The RASCALLI platform is implemented in three layers: An Infrastructure Layer, which contains the basic development tools and libraries; a Framework Layer, comprising the general platform services and components; and an Agent Layer, which is the actual application layer containing the RASCALLI agents (see table 4.2 on the following page). These layers will now be explained on a conceptual level while more detailed information will be provided in the subsequent chapters.

Layer	Description
Agent Layer	Agent architectures, components, definitions and instances
Framework Layer	Technical services and utilities (e.g. networking support, RDF support, agent management)
Infrastructure Layer	Basic tools and components (e.g. Java, Maven, OSGi)

Table 4.2: RASCALLI Platform Layers

4.2.1 Infrastructure Layer

The Infrastructure Layer contains basic tools and components used in the RASCALLI project. Specifically, these are Java, Maven[1] and OSGi[2]. In addition, this layer contains custom-made development and administration tools for the RASCALLI platform, such as user interfaces for agent configuration and deployment tools.

The most important feature of the Infrastructure Layer is the use of OSGi, which implements a dynamic component model on top of Java. This means that components can be installed, started, stopped and uninstalled at runtime. Furthermore, dependencies between components are managed by OSGi in a fashion that allows the execution of multiple versions of a single component at the same time. Finally, OSGi provides a framework for service-based architectures, where each component can provide services to other components, based on Java interface specifications.

The use of OSGi thus enables the platform features *multi-version* and *extensibility*, and supports the implementation of a *component-based architecture* in the upper two platform layers.

4.2.2 Framework Layer

The Framework Layer comprises general platform services and utilities employed by the Rascalli, including agent and user management, communication (user-to-agent, agent-to-agent), event handling, technology integration and various other platform services.

The services on this layer provide the basis for the *multi-agent*, *multi-agent-architecture*, *multi-user* and *communication* features of the platform.

[1] `http://maven.apache.org/`
[2] `http://www.osgi.org/`

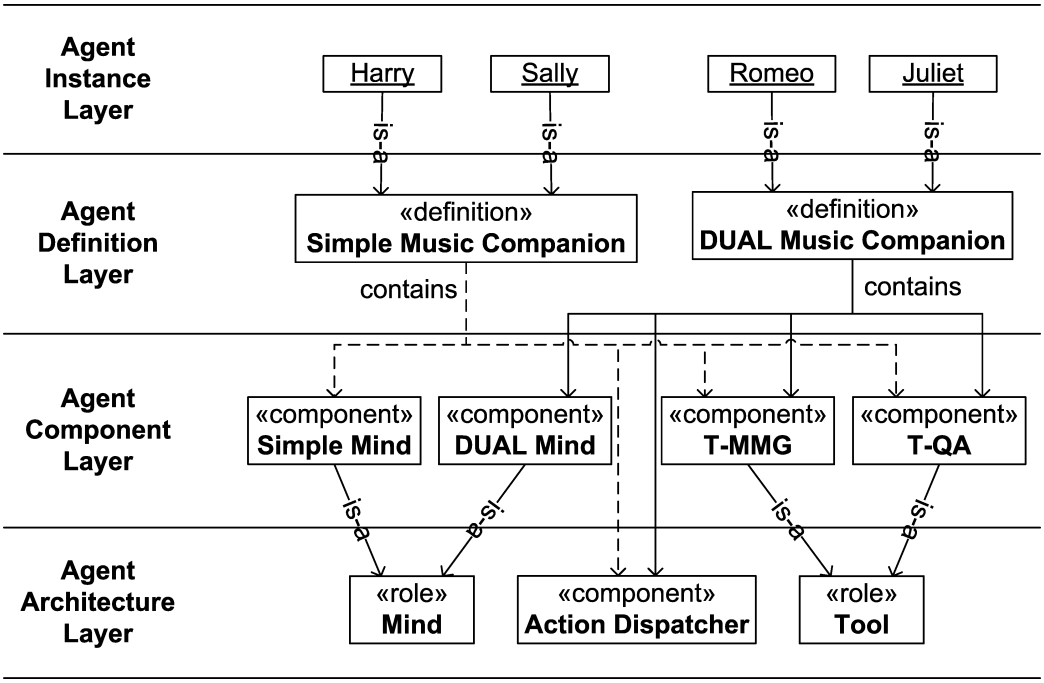

Figure 4.1: The four sub-layers of the Agent Layer, with a few selected components of the Mind-Body-Environment agent architecture (see section 7.2 on page 60).

4.2.3 Agent Layer

The Agent Layer is the application layer of the platform and contains the implementation of the actual agents. It is designed to support the development and execution of multiple agents of different kinds as required by the project objectives. This layer consists of the following sub-layers:

Agent Architecture Layer: An agent architecture is a blueprint defining the architectural core of a particular type of Rascalli. More precisely, it sets the roles of agent components and provides means for defining and assembling a specific agent. The architecture can also contain implementations of common components shared by all agent definitions.

Agent Component Layer: Contains implementations of the roles defined on the Agent Architecture Layer.

Agent Definition Layer: An agent definition is an assembly of specific components of the Agent Component Layer of a specific agent architecture. Different agent definitions for the same agent architecture might contain different components for certain roles.

Agent Instance Layer: Contains the individual agent instances. Each Rascallo is an instantiation of a specific agent definition.

Note that these sub-layers are not technically enforced by the platform, which would be quite impossible. Instead, they are conceptual guidelines which, if adhered to, lead to a modular agent implementation well suited for component re-use and simple assembly of agents.

Figure 4.1 on the previous page gives a (simplified) example of an agent architecture (the Mind-Body-Environment architecture, see section 7.2 on page 60). The Agent Architecture Layer defines two roles, Mind and Tool, and implements an agent component (Action Dispatcher) shared by all agent definitions. The Agent Component Layer contains two implementations of the Mind role, as well as two Tools. Based on this architecture, two agent definitions combine each of the Mind implementations with the available Tools and the Action Dispatcher into different kinds of agents. Finally, a number of agent instances are shown on the Agent Instance Layer.

4.3 Relation to other Agent-Based Systems

The Agent-Based Systems Reference Model (ABSRM, [MMM+06]) defines a general framework for the classification of agent-based software systems. It has been derived from existing agent-based systems (e.g. JADE) via forensic software analysis. This model defines five layers on which agent-based systems or parts of such systems can be mapped in order to allow for a comparison of different systems. The following is an attempt at mapping the RASCALLI platform to the layers of this reference model. Table 4.3 provides an overview of this mapping.

ABSRM Layer	**RASCALLI Platform Layer**
Agents layer	Agent Layer
Framework layer	Framework Layer
Platform layer	Infrastructure Layer
Host layer	n/a
Environment layer	n/a

Table 4.3: Mapping of RASCALLI Platform Layers to ABSRM Layers

Agents Layer:

> The Agents layer consists of agents that perform computation, share knowledge, interact and generally execute behaviors in order to achieve application level functionality. We make few assumptions about the Agent layer except to state that agents are situated computational processes – instantiated programs that sense and effect an environment in which they exist. We make no assumptions about the internal processing structures of an agent. An agent could be built with a complex cognitive model or it could be a simple rule-based system. [MMM+06, page 4]

This definition of the Agents layer is quite consistent with the definition of the RASCALLI platform's Agent Layer. The platform also makes little assumptions as to the implementation of a given agent architecture, with only a few technically motivated exceptions, such as the need for an agent factory (see 6.1.3.1). However, the sub-layers of the RASCALLI platform's Agent Layer function as a guideline for the development of component-based agents.

Framework Layer: The Framework layer provides supporting functionality and services for agents, including agent life-cycle management, agent administration, communication support, etc. This again maps very well to the RASCALLI platform's Framework Layer.

Platform Layer: This layer contains components such as operating systems, middleware (e.g. databases) and other software used to execute an agent-based system. This layer maps to the RASCALLI platform's Infrastructure Layer.

Host Layer: The Host layer is the hardware an agent system runs on. Since the RASCALLI platform is built in Java, it is more or less portable and hardware-agnostic. It should, however, be noted that OSGi has originally been developed for embedded systems and is therefore sufficiently small to be executed on many kinds of devices, such as handheld devices or robot hardware, as long as the operating system supports Java.

Environment Layer: The Environment layer is the physical world in which an agent system exists. Currently, RASCALLI agents live only in the virtual realm of the Internet, but they can interact with real-world entities such as humans. It is also feasible to use the RASCALLI platform for

robotic systems, either by remote-controlling robots or even by embedding the control software in the robot. However, in the latter case, the currently implemented services and components would probably be of little use.

4.4 Summary

In this chapter, we have described the primary features of the RASCALLI platform. We have also introduced the software layers of the platform architecture:

- the Agent Layer (with four sublayers),
- the Framework Layer, and
- the Infrastructure Layer.

Finally, we have shown the relation to other agent frameworks by mapping the RASCALLI platform layers to the Agent-Based Systems Reference Model.

Chapter 5

Infrastructure Layer

The Infrastructure Layer comprises the software components that form the basis of the RASCALLI platform, including Java, Maven and OSGi. Furthermore, it contains a basic development environment, building on those technologies, that allows for the creation and management of agents and agent components for and within the platform.

5.1 Technology Overview

The RASCALLI platform is built upon a number of tools and technologies, including Java, Maven and OSGi. This section provides a more detailed description of the latter two and also explains how they are used within the RASCALLI context. While it is assumed that Java is sufficiently known to make a detailed description superfluous, the following section motivates the decision to use Java instead of any other programming language.

5.1.1 Why Java?

The decision for Java was based on the following considerations:

Libraries: A huge number of software libraries and APIs is available for Java. While this is also true for many other languages, we can at least assume that other languages probably do not have better library support than Java.

Platform independence: Even though this argument has probably been overused, it is worth mentioning that the JVM is available for many operating systems and Java applications are therefore mostly platform independent.

Excellent development support: Some of the best integrated development environments (IDEs) are for the Java language, especially when it comes to freely available and open source applications, such as Eclipse[1]. The availability of low-cost or even no-cost software is obviously important in a research context.

In addition to powerful and free IDEs, the Java community also boasts other excellent development tools, such as build systems like Maven and Ant[2], test frameworks, continuous integration frameworks, and many more.

OSGi: The availability of a dynamic component system like OSGi (see below) is quite unique. While it is, of course, possible to load components into a software system dynamically with almost any current programming language, the support for multiple versions of a component and the dependency management provided by OSGi is probably hard to find in other systems.

Dynamic language integration: While Java is a statically typed, compiled language, which may be considered inappropriate for a research context, many dynamic languages have been integrated with or even specifically created for the JVM. This includes wide-spread languages, such as Ruby (JRuby[3]), Python (Jython[4]) or Groovy[5], and even functional languages like Scala[6]. It is also quite easy to invoke programs written in other languages, such as C or Perl, from a Java application.

The availability and excellent integration of dynamic languages opens the path towards an interesting scenario, where the platform is implemented in Java and individual agents are then implemented in Groovy, for example. This allows for a good mix of performance and stability on the one hand, and flexibility and implementation speed on the other hand.

Support for DSL: Java (and some of the dynamic languages for the JVM) offer excellent support for Domain Specific Languages (DSL, [MHS05]). In a DSL scenario, agents might be implemented in such a way that they can then be assembled using a special language (a DSL).

[1] `http://www.eclipse.org/`
[2] `http://ant.apache.org/`
[3] `http://jruby.codehaus.org/`
[4] `http://www.jython.org/`
[5] `http://groovy.codehaus.org/`
[6] `http://www.scala-lang.org/`

5.1.2 Maven

Maven[7] is an open-source tool for software project management. It can manage a project's build, deployment, reporting, documentation and more. All of these activities are based on a single configuration file (POM), which contains, among other things, the project's name and version, its dependencies on other projects, and configuration parameters for the various Maven components. In addition to this configuration, Maven builds on the notion of *convention over configuration*, meaning for example that a project's subdirectories should be laid out in a certain fashion (*convention*) rather than being configured independently for each project. This approach has a number of advantages, but a detailed discussion is beyond the scope of this document.

In most cases, a Maven project produces what is called a *Maven artifact*. Essentially, this is a software bundle containing the compiled software and configuration. This artifact, together with the project's POM file, is deployed to a Maven repository. When a project specifies a dependency on other projects in its POM, the corresponding artifacts are automatically downloaded and integrated by Maven. This sort of automatic dependency management greatly facilitates the implementation of software systems that depend on many external components. It is therefore a very good fit for a component-based software architecture, where each component can be implemented as a separate Maven project.

In the RASCALLI context, all platform components are implemented as Maven projects. We also make use of Maven's OSGi support, which allows us to generate OSGi bundles from our component projects.

5.1.3 OSGi

> The OSGi technology is a set of specifications that define a dynamic component system for Java. These specifications enable a development model where applications are (dynamically) composed of many different (reusable) components. [OSG08b]

5.1.3.1 Motivation

By the definition of the OSGi Alliance[8], OSGi is "The Dynamic Module System for Java". [OSG08a] contains a long list of advantages of using OSGi. Some of the more important ones (in the context of the RASCALLI platform) are:

[7] `http://maven.apache.org/`
[8] `http://www.osgi.org/`

Reuse: "The OSGi component model makes it very easy to use many third party components in an application. An increasing number of open source projects provide their JARs ready made for OSGi. However, commercial libraries are also becoming available as ready made bundles." [OSG08a]

In addition to this, it also allows for easy integration of external non-OSGi or even non-Java components, because they can be hidden behind an OSGi service and made available to other components in the platform.

Dynamic Updates: "The OSGi component model is a dynamic model. Bundles can be installed, started, stopped, updated, and uninstalled without bringing down the whole system." [OSG08a]

This is especially useful in the context of a platform shared between multiple developers. Individual components can be added or updated while the rest of the system continues running.

Versioning: "OSGi technology solves JAR hell. JAR hell is the problem that library A works with library B;version=2, but library C can only work with B;version=3. In standard Java, you're out of luck. In the OSGi environment, all bundles are carefully versioned and only bundles that can collaborate are wired together in the same class space. This allows both bundle A and C to function with their own library." [OSG08a]

This is, again, very useful or even necessary for a shared development platform.

Widely Used: "The OSGi specifications started out in the embedded home automation market but since 1998 they have been extensively used in many industries: automotive, mobile telephony, industrial automation, gateways & routers, private branch exchanges, fixed line telephony, and many more. Since 2003, the highly popular Eclipse Integrated Development Environment runs on OSGi technology and provides extensive support for bundle development. In the last few years, OSGi has been taken up by the enterprise developers. Eclipse developers discovered the power of OSGi technology but also the Spring Framework helped popularize this technology by creating a specific extension for OSGi. Today, you can find OSGi technology at the foundation of IBM Websphere, SpringSource Application Server, Oracle (formerly BEA) Weblogic, Sun's GlassFish, and Redhat's JBoss." [OSG08a]

OSGi has been one of the major buzz-words in the Java community for the last couple of years. Since one of the goals of the RASCALLI platform is to bring state-of-the-art development tools and methodologies to agent development, it seems to make sense to use a technology that has been so widely adapted.

5.1.3.2 OSGi Architecture

OSGi is built around the concept of a *bundle*. In OSGi terminology, a bundle is a component that can be installed, started, stopped, updated and uninstalled in an OSGi runtime environment (*container*) independently from any other bundles. A bundle can provide two kinds of services to other bundles:

- It can export Java packages for other bundles to use. These bundles can then instantiate classes implemented in the exported packages.
- It can register services with the OSGi service registry, where other bundles can find and use them. Services are defined with Java interfaces rather than a special interface specification language. These interfaces must be exported in the bundle's Java packages so that other bundles can use them.

A bundle does not declare its dependencies on other bundles directly (by naming a specific bundle), but rather by importing the Java packages it needs. OSGi automatically resolves these dependencies and builds a class path for each bundle, containing any required classes. The client bundle is ignorant of which specific bundles provide the required packages.

5.1.3.3 OSGi Implementations

There are currently three major open-source implementations of the OSGi specifications:

- Apache Felix[9],
- Eclipse Equinox[10], and
- Knopflerfish[11].

[9] `http://felix.apache.org/`
[10] `http://www.eclipse.org/equinox/`
[11] `http://www.knopflerfish.org/`

The RASCALLI platform has been developed on Apache Felix, but this was a rather arbitrary choice. Also, since these are all implementations of the current OSGi standard, it should be rather effortless to port the platform to any of the other containers. Only a very small part of the platform source code is Felix specific (namely a set of Felix shell commands).

5.1.3.4 Convenience Layers on Top of OSGi

While OSGi provides a very simple and easy to use service framework, there are several issues with implementing an application directly on the service specifications of OSGi. For example, it is quite tedious to track the availability of service instances. There are currently two major efforts in the open source community to ease these problems by providing a framework that manages OSGi services and components, and their dependencies upon each other. These are:

- Apache iPOJO[12] and
- Spring Dynamic Modules[13].

Apache iPOJO [EHL07] is a runtime environment for service components. Components are implemented as simple Java objects (POJO means Plain Old Java Object), and dependencies on other components as well as component properties are injected at runtime. The definition of an iPOJO component can be done either in XML metadata or through the use of Java annotations. Component instances are defined in an XML metadata file and automatically instantiated by iPOJO when the bundle is loaded into the OSGi container. If a component provides a service, iPOJO registers the component with the OSGi service registry.

Service dependencies are injected either directly into Java instance variables or into methods of the Java object. iPOJO manages the entire dependency tree, so that individual components are either valid or invalid, depending on whether their dependencies can currently be satisfied with the services available in the OSGi container.

This framework makes the implementation of service-based OSGi application really simple, but there are some limitations. For example, if component instances need to be created at runtime, this cannot easily be done with iPOJO, even though iPOJO supports the concept of OSGi service factories. For this reason, the RASCALLI Framework (chapter 6 on page 44) has the

[12] `http://felix.apache.org/site/ipojo.html`
[13] `http://www.springframework.org/osgi`

concept of Agent Factories, which are explicitly implemented in Java instead of using OSGi or iPOJO mechanisms.

Spring Dynamic Modules (Spring DM) is an extension of the Spring Framework[14] for OSGi. OSGi services and dependencies can be configured as Spring beans in XML or any other variant of Spring configuration. At the beginning of the development of the RASCALLI platform, the Spring Dynamic Modules project was still very much at the beginning. Therefore, all currently available platform components are based on iPOJO. However, since both frameworks eventually use the OSGi service framework, it is possible to mix bundles based on either of these technologies within a single OSGi container. Due to the fact that the Spring Framework is a widely accepted technology in the Java community and that Spring DM seems to advance more rapidly than iPOJO, it might be advisable to switch to Spring DM in the future.

5.2 Development Environment

The RASCALLI platform is both a runtime and a development environment for software agents. While the runtime part is fully functional at the time of writing, the development parts have been implemented only as far as necessary for the implementation of the RASCALLI agents. This includes the following features:

Maven environment: Based on a parent POM which contains the necessary plug-ins and configuration, a Maven project environment has been implemented. Platform sub-projects inherit from this parent project and can then be built for and deployed to the RASCALLI Maven repository.

Platform configuration: The platform configuration and startup scripts have been designed in such a way that multiple different platform environments, with different bundle configurations can be executed. While this is in sharp contrast to the *shared platform* feature, it is a necessary intermediate step until a full-fledged development environment is available. The problem is that currently only a single-user command shell is available for OSGi management, so that developers could theoretically share a single platform, but only one developer at a time can manage OSGi bundles.

[14] `http://www.springframework.org/`

Felix command shell extensions: The Felix command shell is an administration interface for the Felix OSGi container and provides basic OSGi bundle management functionality. Several additional commands have been implemented for this shell.

Possible improvements and extensions to the development environment are discussed in chapter 9 on page 74. Most notably, a multi-user management tool will be necessary, for example a web-based user interface for bundle and agent management.

5.3 Summary

The RASCALLI platform is based on Java, Maven and OSGi. The use of Java is motivated by the availability of OSGi and other important features. Platform and agent components are implemented as separate Maven projects, which are then compiled into OSGi bundles and deployed to the platform. OSGi is the technological foundation for many of the RASCALLI platform features, including its *extensibility*, support for *multiple versions* of each component and the *shared platform* concept.

Chapter 6

Framework Layer

This chapter contains details about the services provided by the Framework Layer and their implementation. Even though this layer is implemented entirely in Java, the use of Java terminology is mostly avoided. Instead, we speak of services and components, which are denoted with upper case first letters (e.g. *Agent Manager*). Note that the term *Agent*, with an upper case first letter, denotes a software component, as opposed to the more general use of lowercase *agent* throughout the rest of the document.

6.1 Agent Management

From the platform's point of view, *Agent* is a very generic concept. An Agent is a specific instance of an Agent Definition, and has a defined state and behavior. The Agent Manager service is responsible for loading Agent State from external persistent data storage, creating the Agents, starting and stopping them, and updating the Agents' state if it changes in the external data storage (e.g. if a user changes an Agent's configuration).

6.1.1 Agent State

Agent Configuration: This is persistent data that specifies a particular Agent. It includes the following attributes:

- A unique (within the platform instance) Agent ID,
- a User ID, specifying the Agent's owner,
- an Agent Definition ID, specifying the Agent Definition to be used for instantiating this Agent, and

- a set of properties consisting of key-value pairs. The exact set of properties is defined by the Agent Definition.

The Agent Configuration is stored in the Agent Modeling Server (AMS)[1]. Updates to the Agent Configuration are pushed to the Agent by the Agent Manager.

Agent Profile: This is persistent data generated during the Agent's lifetime. This includes training data gathered during the training sessions with the user or interaction data collected from other user or agent interactions, as well as any information learned during the Agent's life time (the exact nature of this data depends on the agent architecture and Agent Definition, so it cannot be specified in more detail here). The Agent Profile is also stored in the Agent Modeling Server (again, this is not a technical requirement).

In contrast to the Agent Configuration, the Agent Profile is not pushed into the Agent by the Agent Manager. Instead, the Agent is supposed to actively access its profile in order to fulfill its tasks (e.g. retrieving keywords for RSS news filtering).

Runtime state: This is state the Agent keeps in memory while it is running. When the Agent is stopped, the runtime state is lost.

6.1.2 Agent Life-cycle

The following events occur during an Agent's life-cycle (see Figure 6.1 on the next page):

- **Creation:** The Agent is created by an Agent Factory.
- **Start:** The Agent Manager starts the Agent.
- **Update:** The Agent Configuration has changed.
- **Stop:** The Agent Manager stops the Agent.
- **Destruction:** The Agent is consumed by the JVM's garbage collector.

[1]The AMS is not part of the RASCALLI platform. It is a service provided by project partner SAT and is currently used to store Agent Configurations and Agent Profiles. Note that there is no technical requirement to use the Agent Modeling Server for storing persistent Agent data. An agent architecture can make use of arbitrary data stores for persistent state. Also, the Agent Manager (see section 6.1.3 on page 47) is designed to load Agent Configurations from arbitrary sources.

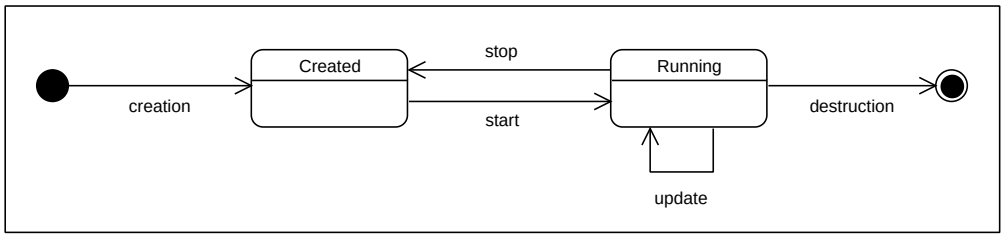

Figure 6.1: Agent Life-Cycle

6.1.2.1 Event details

Creation: An Agent is created by an Agent Factory (see section 6.1.3.1 on page 48). Since the Agent Factory is part of an Agent Definition, the exact mechanism of Agent creation cannot be defined at this point. However, the Agent Manager passes an Agent Configuration to the Agent Factory which then builds the Agent. Typically, the Agent is assembled from a set of Agent Components. It will also get access to its Agent Configuration.

Start: When the Agent is started, it is expected to do any of the following:

- Allocate resources, such as database connections,
- start threads, and
- load persistent state.

Update: Whenever the Agent Configuration for an Agent changes, the Agent Manager will pass the new configuration to the Agent. The Agent is then expected to adapt its internal state to the new configuration. For example, the user might add another RSS feed to an Agent's list of feeds to be monitored. In this case, the Agent is expected to start monitoring the new feed.

Stop: When an Agent is stopped, it is expected to do any of the following:

- De-allocate resources (e.g. release database connections),
- terminate threads, and
- store persistent state.

Destruction: When an Agent Configuration is deleted (for reasons for deleting an Agent see section 6.1.3), the Agent Manager will stop it (if it is running) and then release it to the JVM's garbage collector. The Agent will then be consumed by the garbage collector at an arbitrary time.

6.1.2.2 Extended Agent Life-Cycle

The Agent life-cycle detailed above is that of an actual Agent instance running in the platform. In the larger context of the entire RASCALLI system, including persistent storage, an agent (or rather an agent's persistent state) has a much longer life-cycle:

- **Creation**: The user creates the Agent Configuration.
- **Update**: The user updates the Agent Configuration (via a configuration interface) or the Agent Profile (via interaction with the agent).
- **Deletion**: The user deletes the Agent Configuration.

Between these events, the agent life-cycle specified above might be executed multiple times (e.g. each time the platform is started/stopped, the Agent is created/destroyed).

The persistent agent state (configuration and profile) must be stored during the entire life-cycle, so that it is available when the Agent is restarted.

6.1.3 Agent Manager

The Agent Manager is responsible for creating, starting, updating and stopping Agents running in the platform. In order to fulfill its task, the Agent Manager makes use of several other platform components. Figure 6.2 on the following page shows an example of the collaboration between the Agent Manager and one instance of Agent, Agent Factory and Agent Configuration Source, respectively. In this example, the Agent Configuration Source detects a newly created Agent Configuration (x) in the persistent storage (e.g. a config file). It sends the new Agent Configuration to the Agent Manager, which looks up the appropriate Agent Factory and tells it to create a new Agent, based on the Agent Configuration. The factory then creates the Agent (a). Later, the user changes the configuration, which is detected by the Agent Configuration Source. The updated Agent Configuration (x') is sent to the Agent Manager, which passes it on to the previously created Agent.

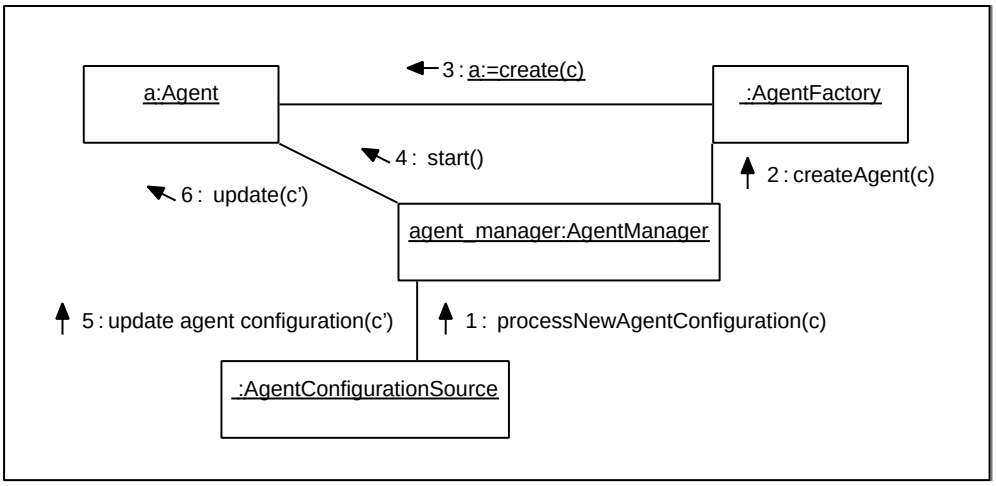

Figure 6.2: Example of the collaboration between the Agent Manager and other components

6.1.3.1 Agent Manager Collaborators

Agent Configuration Source: An Agent Configuration Source manages a set of Agent Configurations in an external persistent storage. This might be a configuration file, a database or, as is the default case for the RASCALLI platform, the Agent Modeling Server.

Agent Definition: The concept of agent definitions has already been described in section 4.2.3 on page 32. On the implementation level, an Agent Definition is an OSGi bundle that contains an Agent Factory.

Agent Factory: Each Agent Definition contains an Agent Factory. This component is responsible for creating Agents of this Agent Definition.

6.1.3.2 Agent Manager Life-cycle

This section gives a more complete view of this collaboration and the resulting behavior of the Agent Manager, based on events that can happen during the Agent Manager's life-cycle.

The Agent Manager is started: This usually happens when the platform is started, but can also happen if the bundle containing the Agent Manager has been updated. At this point, the Agent Manager knows nothing about existing Agent Definitions or Agent Configuration Sources.

The Agent Manager is stopped: This can happen when the platform is stopped or when the bundle containing the Agent Manager is stopped or updated. The Agent Manager stops all running Agents and releases them for garbage collection.

An Agent Configuration Source informs the Agent Manager that a new Agent Configuration has been created: This might happen when a user creates a new agent, but also when an Agent Configuration Source is first started in the platform. The Agent Manager extracts the Agent Definition ID from the new Agent Configuration. If an Agent Definition exists for a given agent, the Agent Manager passes the Agent Configuration to the appropriate Agent Factory in order to create the Agent. Afterwards, the new Agent is started. If the Agent Definition does not exist, the Agent Configuration is stored locally in case the Agent Definition is added to the platform later on.

An Agent Configuration Source informs the Agent Manager that an Agent Configuration has been updated: This happens when a user changes the agent's configuration in the external data store. The Agent Manager passes the new configuration to the respective Agent. If the Agent does not exist in the platform, it is created (see previous event).

An Agent Configuration Source informs the Agent Manager that an Agent Configuration has been deleted: This happens when a user deletes one of her agents. The Agent Manager stops the Agent (if it is running) and releases it for garbage collection.

An Agent Definition registers its Agent Factory with the Agent Manager: This happens whenever an Agent Definition bundle is started in the RASCALLI platform (e.g. at startup or when a new Agent Definition is being deployed). The Agent Manager looks through all available Agent Configurations in order to find those for the newly registered Agent Definition (note that the Agent Definition ID is part of the Agent Configuration). If it finds any such configurations, they are passed to the Agent Factory, which creates the new Agents. Afterwards, the Agent Manager starts these Agents.

An Agent Definition unregisters its Agent Factory from the Agent Manager: This happens when an Agent Definition is removed from the platform or the platform is stopped. The Agent Manager stops all Agents for that Agent Definition and releases them for garbage collection. The Agent

Configurations are kept in the local storage in case the Agent Definition is re-added to the platform later.

6.1.3.3 Sample Agent Configuration Sources

AMS-based Agent Configuration Source: This configuration source retrieves Agent Configurations from the Agent Modeling Server (see 6.1.1). The configurations are created, updated and deleted via the RASCALLI web interface (see section 8.1.2 on page 67). The Agent Configuration Source polls the AMS at regular intervals for updated Agent Configurations and passes them on to the Agent Manager. The interaction between the Agent Configuration Source and the Agent Modeling Server is based on a web service interface.

File-based Agent Configuration Source A configuration source that reads Agent Configurations from a configuration file. It scans the file regularly to detect changes.

6.2 User Management

User management is implemented in a similar way to agent management. A User Service keeps a list of all users. Users can be added, modified and removed by User Configuration Sources in the same way Agent Configurations are supplied by Agent Configuration Sources. Currently, the only User Configuration Source available retrieves users from the Agent Modeling Server. User accounts can be created via the Web User Interface (see section 8.1.2 on page 67).

The service provided by the User Manager is, however, much simpler than that of the Agent Manager. Users can be retrieved by ID and can be authenticated with user name and password.

6.3 Event Handling

The RASCALLI platform makes use of an Event Service to dispatch asynchronous events between components running inside the platform. Following the criteria proposed by [Elh99], the current implementation can be characterized as a synchronous push model. Event listeners are organized in a hierarchical way, with the Event Service as the root node.

For example, each Agent is automatically registered as an Event Listener with the Event Service after it has been created. It is up to each specific

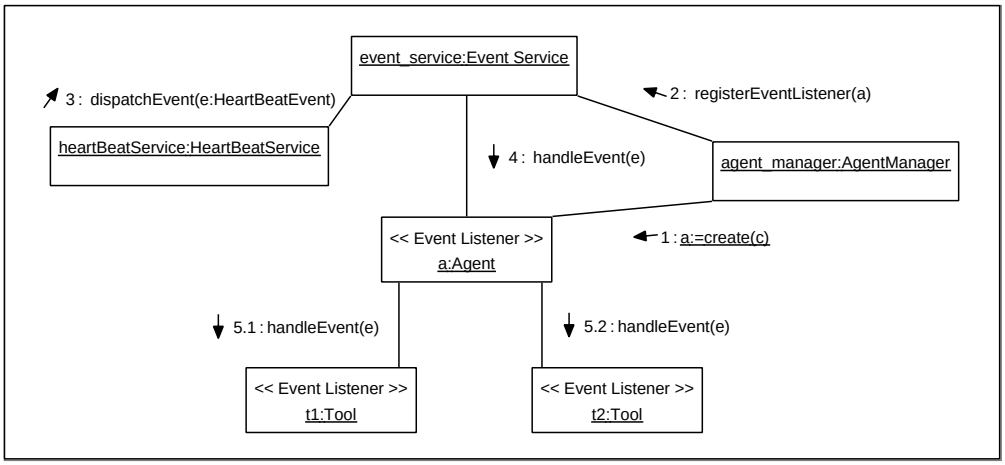

Figure 6.3: Example of the collaboration between the Event Service and other components

agent architecture to implement further event handling within the Agents. The MBE agent architecture (see section 7.2 on page 60) allows each Agent Component to be an Event Listener, so that the components receive all Agent Events.

Figure 6.3 shows a sample interaction between the Event Service and some other components: After creating an Agent instance, the Agent Manager registers the Agent as an Event Listener. When the Heart Beat Service dispatches a Heart Beat Event to the Event Service, this Event is forwarded to the Agent, which in turn forwards it to all interested Agent Components.

Events can be objects of any type and it is up to the receiver of an event to decide whether and how to process it. Therefore, the event system itself is untyped in the sense that any kind of event can be sent across a single communication channel. On the other hand, the event type can be determined and the event then be processed in a type-safe manner.

It is possible to inject an event at any point in the hierarchy. For example, when a user connects to an Agent, the Agent receives an appropriate event. Since this event is of no interest to all the other Agents, it is sent directly to the Agent, instead of to the Event Service.

A more sophisticated approach could improve the performance and robustness of the event system by introducing event channels and asynchronous event dispatching.

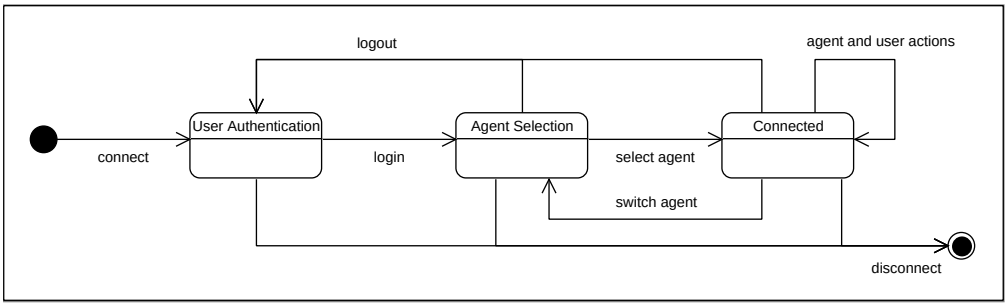

Figure 6.4: Communication protocol states between platform and 3D client

6.4 Communication

The RASCALLI project objectives explicitly contain requirements for agent-to-user and agent-to-agent communication. While the former is obvious, the latter results from the wish to create communities of specialized agents, which interact in order to improve each agent's performance.

Due to the platform's extensibility, it would be easy to add new communication channels, but this section focuses on the ones already implemented in the current system.

6.4.1 Agent-to-user Communication

6.4.1.1 3D Client Integration

The 3D client[2] is one of the major interaction interfaces between user and agent. It displays the agent as a 3-dimensional character and produces synthesized speech as well as gesture output from the agent's messages to its user. User input is restricted to typed text input and a small number of buttons (speech recognition was beyond the project scope). See section 8.1.3 on page 68 for more details about the 3D client.

In order to facilitate this kind of interaction, a TCP-based communication service has been implemented in the RASCALLI framework. This service is designed to support arbitrary communication protocols, making it easy to integrate additional user interfaces in the future.

The communication protocol for the 3D client integration is based on the exchange of XML messages and a simple finite state machine, which is depicted in figure 6.4:

[2]Provided by project partner Radon Labs.

1. After connecting to the platform, the client must send the user's login data (username and password).

2. If the user is correctly authenticated, the platform sends back the list of agents available to the user. The user must then select one of the agents. After this, the user is connected to the selected agent and can communicate with it.

3. The user can switch agents or logout at any time.

4. While connected to an agent, the following user actions are sent to the platform:

   - User utterances (text input),
   - praise and
   - scolding for the agent, and
   - other user actions, such as clicking Web links sent to the user or switching to other user interfaces (see chapter 8 on page 66 for available interfaces).

5. At the same time, the agent can send messages to the user. These are encoded in an XML format that includes SSML (Speech Synthesis Markup Language)[3] and BML (Behavior Markup Language, [V+07]). The 3D client parses these messages, generates text and speech output, and animates the 3D character.

When a user message arrives in the platform, it is translated into an Event and dispatched to the Agent via the Event System. The following types of events are currently implemented:

- User Connected,
- User Disconnected,
- User Utterance (text input),
- User Praise,
- User Scolding,
- User Context Change (the user switched to another user interface), and

[3] `http://www.w3.org/TR/speech-synthesis/`

- User URL Click (the user followed one of the Web links sent by the agent).

It is up to the agent architecture to deal with these Events in an appropriate fashion. Examples include answering a question posed by the user (User Utterance) or implementing some sort of learning strategy based on User Praise or Scolding.

6.4.1.2 Jabber Integration

Since Rascalli are supposed to be independent entities living on the Internet, it makes sense to equip them with widespread Internet communication tools. One of these tools is instant messaging, which has been integrated with the RASCALLI platform employing Jabber. Jabber[4] is an open instant messaging technology based on the Extensible Messaging and Presence Protocol (XMPP)[5] standard.

The Framework Layer contains a Jabber client implementation[6]. This can be used by an agent architecture, if it wants to make use of Jabber communication. If it is used, a Jabber ID is automatically created on a dedicated Jabber server for each new agent. If the agent's user has specified a Jabber ID in its profile, the user is automatically added to the agent's contact list and thus a connection between agent and user is established. When the user is online in Jabber, the agent can send text messages to the user's Jabber client.

The implemented Jabber client makes use of the Event System to inform the Agent about the following Events:

- Jabber Presence Changed (the agent's user has gone online or offline), and
- User Utterance (a text message from the user).

On top of this basic mechanism, an agent architecture can implement a custom communication protocol. For example, a simple command language could be implemented in place of or in addition to natural language text input.

[4] `http://www.jabber.org/`

[5] XMPP Standards Foundation (`http://www.xmpp.org/`)

[6] The initial implementation of the Jabber client was provided by project partner OFAI and has then been adapted and integrated with the RASCALLI framework.

6.4.1.3 Web interface

As an inhabitant of the Internet, RASCALLI agents naturally have a web presence, which is part of the Web User Interface[7] (for details see section 8.1.2). From the user's point of view, this interface is used for the following purposes:

- Creating a user account,
- creating and configuring agents, and
- selecting agents for creating and updating its profile via one of the music browsing interfaces[8] (see chapter 8 on page 66).

The agent, on the other hand, uses the web interface to post information for the user via a web service interface[9]. This includes:

- Answers to questions asked by the user in one of the interactive interfaces (3D client or Jabber). This is especially useful if the answer is too long or otherwise unsuitable for speech synthesis. In such a case, a short answer is given at the interactive interface, together with a link to the longer answer that was posted to the web interface.
- Information found on behalf of the user, such as interesting RSS feed items. In this case, the items are posted on the web interface and the user is informed that new information is available (if the user is currently online), again with a link to the posting.

6.4.2 Agent-to-agent Communication

At the current state, agent-to-agent communication has not yet been implemented. However, in the final version of RASCALLI, an agent will be able to use the Agent Modeling Server to find relevant agents for solving a given task. For example, an agent might try to improve its RSS filtering capabilities by looking for agents with similar agent profiles. The agent can then communicate with those other agents and exchange RSS feed URLs or filter keywords.

[7]The Web User Interface is provided by project partner SAT.

[8]The browsing interfaces have been provided by project partners SAT (Music Explorer) and DFKI (Visual Browser).

[9]Actually a part of the web service interface of the Agent Modeling Server, which is also provided by SAT.

The implementation of agent-to-agent communication in the Framework Layer is rather straightforward. We envision three scenarios that could be easily implemented based on the existing framework components:

Agent-to-agent communication on the Java level: Java method invocations are a viable option for agent communication, since all agents exist within a single Java virtual machine. This approach could be implemented as an extension of the Agent interface or preferably by using the existing Event System. In the latter case, an Agent could send an Event to another Agent, containing an arbitrary message.

Agent-to-agent communication via a blackboard service: Via a blackboard service implemented in the Framework Layer, agents could broadcast messages or send messages to other agents.

Agent-to-agent communication with Jabber: Since all agents have a Jabber account, they might as well use it to communicate with each other. The advantage of this approach is that it would transparently allow for future distributed versions of the platform, where agents communicate with each other across the Internet.

Based on these basic mechanisms, which would be implemented in the Framework Layer, an agent architecture might choose any kind of communication protocol between its Agents. For example, Agents might exchange plain text or custom XML messages, or they might make use of an established agent communication language.

6.5 Other Services and Components

In addition to the services already presented, the Framework Layer contains a number of useful components to be used by the Agent Layer.

6.5.1 Configuration Management

The RASCALLI platform provides a single location where all components deployed to the platform can store their configuration. The idea is that each component that needs configuration creates directories and files below a specified root directory.

The Configuration Service provides a single point of access to this configuration directory by resolving relative file system paths to absolute paths.

6.5.2 RSS Feed Management

The RSS Manager is an example of the benefits of having a single runtime environment. Each of the currently implemented agents is capable of monitoring RSS[10] feeds, filtering their contents and passing interesting items on to their users. In this situation, it is quite likely that multiple agents will access the same RSS feeds. The RSS Manager is a service component in the Framework Layer that allows agents to subscribe to RSS feeds. It monitors these feeds and passes new items on to the subscribed agents.

6.5.3 Utility Components

The Framework also contains a number of smaller components that implement useful functionality, such as:

Heart Beat: A component that generates a Heart Beat Event at regular intervals, so that other components can easily track time.

Thread Pool: This component wraps a Java thread pool as an OSGi service.

Class Loader: A utility to resolve OSGi classloading issues.

System Command Execution: A utility for the execution of native system commands.

6.6 Summary

In this chapter, we have introduced the major services and components of the Framework Layer. These are major platform services for:

- Agent Management,
- User Management,
- Event Handling and
- Communication.

[10] Really Simple Syndication, see RSS Advisory Board (`http://www.rssboard.org/`)

In addition to these major platform services, the Framework Layer also contains smaller services and components that make the implementation of Agent Layer functionalities easier or provide integration of external services and applications. For example, the web service interface of the Agent Modeling Server has been integrated as an OSGi service.

The current state of the Framework Layer reflects the requirements of the RASCALLI system up to date. Due to its modular OSGi-based implementation, it will be easy to extend the functionality provided by this layer in the future.

Chapter 7

Agent Layer

Two agent architectures have been implemented so far: The Mind-Body-Environment (MBE) architecture and an agent architecture for 3D client integration tests. We start this chapter with a brief introduction of the test agents, while the remainder of the chapter provides a more detailed description of the MBE architecture.

7.1 3D Client Test Agents

In order to be able to systematically test the 3D client integration, a special agent architecture has been implemented. It is rather simple – in fact it consists of only two Java classes: The agent implementation and an Agent Factory. Therefore, it does not serve well as an example of a full-fledged agent architecture, but it can be seen as a proof that multiple agent architectures can actually be implemented in the platform.

The test agents are based on an XML configuration file that defines a number of input/output pairs. When the user sends a defined input to the agent, the agent responds with the corresponding output. Of course, multiple test agents can be created in a single platform instance and the behavior of each agent can be defined in a separate configuration file.

In addition to 3D client integration tests, these agents can also be used for scripted user evaluations, where users must enter a predefined sequence of text messages.

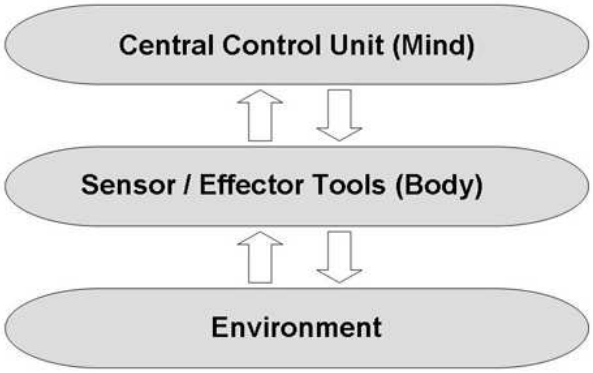

Figure 7.1: A high-level view of the Mind-Body-Environment agent architecture.

7.2 The Mind-Body-Environment Architecture

The agents implemented in the RASCALLI project are based on what we call the Mind-Body-Environment (MBE) agent architecture. Figure 7.1 presents a high-level view of this architecture: An MBE Agent consist of a central control unit (mind), and a set of sensor and effector tools (body), which serve as the mind's interface to the environment. The environment consists of services available on the Internet (e.g. search engines or web services), domain-specific databases, human users, other RASCALLI agents and generally any resource or service accessible from the computer executing the agent.

Communication between mind and tools is based on a shared ontology, describing tools, actions and sensory data so that the mind can interpret messages received from its sensor tools (e.g. a question asked by the user or the result of a database query), select an appropriate action (e.g. respond to the user or access a database), and instruct one of its effector tools to execute the selected action, by sending the action parameters to the tool (e.g. an answer to the user's question or a database query).

The RASCALLI project consortium has agreed on this architecture for the following reasons:

- The basic human-like metaphor of an entity comprising a mind and a body allows for arbitrarily complex implementations of the mind, which can make use of a growing set of tools to perform their tasks on the Internet. The mind implementations can be simple rule-based

action selection or complex cognitive architectures, including cognitive aspects such as learning and planning.

- New tools can be added easily by
  - implementing or integrating them with the platform,
  - extending the shared ontology, and
  - extending the mind to actually use the new tool.
- The simple architecture with only two roles (mind and tool) facilitates integration testing, because each component can be easily replaced with a dummy implementation. For example, a dummy mind can be implemented to test a set of tools even when the actual mind implementation is not yet able to work with them[1].

The following sections describe the implementation of this agent architecture within the RASCALLI platform.

7.3 MBE Agent Architecture Layer

The Architecture Layer defines the basic roles for components of the Agent Component Layer, namely Mind and Tool. While the Java interfaces for these roles are very simple (the Mind must be able to process input data and a Tool must be able to execute an action, based on some sort of action data), the data sent across these interfaces can be arbitrarily complex and is encoded as RDF[2] graphs. These are based on the internal ontology shared between Mind and Tools.

In addition to the basic roles (interfaces), the architecture also implements a Java class MBEAgent (see figure 7.2 on the next page). This class serves as a container for a Mind and one or more Tools to form an actual agent. A component called Action Dispatcher is used by the MBEAgent to forward actions issued by the Mind to the appropriate tools. Finally, the MBEAgent provides access to various platform services to the agent components (e.g. the Configuration Service, the Event Service, the Jabber integration and the RSS Manager).

The architecture layer also provides extensive support for RDF handling so that data exchange between Mind and Tools can be implemented in a straight-forward manner.

[1] This is the reason why the Simple Mind (see below) was initially implemented.

[2] `http://www.w3.org/RDF/`

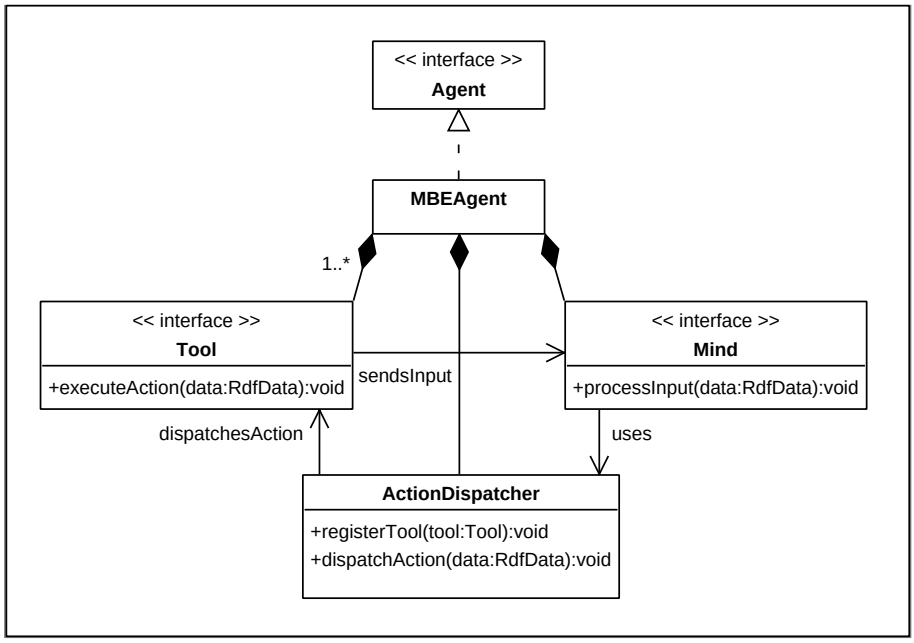

Figure 7.2: MBE Agent class diagram

7.4 MBE Agent Component Layer

This section describes some of the Mind and Tool implementations used in the RASCALLI agents. They form the basis for the MBE agent definitions described in the next section.

7.4.1 MBE Tools

Each of the following Tools[3] is implemented as a separate OSGi bundle. An MBEAgent can use arbitrary subsets of these Tools, depending on the capabilities of the Mind implementation (after all, the Mind must be able to use the selected Tools).

T-IP4DUAL, T-IP4Simple and T-IP4Adaptive: Tools for input processing transform natural language and user feedback inputs into categorized information usable by the respective Mind components (see below).

[3]The Tools have been implemented by various project partners:
T-IP4DUAL, T-IP4Simple, T-IP4Adaptive, T-MMG, T-QA, T-NALQI, T-ChatBot: OFAI; T-QuestionAnalysis: DFKI; T-RSS: OFAI, SAT; T-TextRelevance: SAT.

T-MMG: The Multi-Modal Generation Tool transforms the agent's output from the internal RDF-based representation to the appropriate formats for the interactive user interfaces. For the 3D client, it produces XML output encoding gesture and speech information, and for the Jabber integration, it produces plain text.

T-QA: A general purpose open-domain question answering system based on the work described in [SA05] and [Sko05]. It is used in the RASCALLI platform to provide answers to those user questions that cannot be answered by any of the domain-specific tools.

T-NALQI: A natural language database query interface. The Tool is used in the RASCALLI platform for querying the databases accessible to the Rascalli, in a search for instances and concepts that can provide answers to the user questions. The component analyses natural language questions posed by the user and retrieves answers from the system's domain-specific databases.

T-ChatBot: This is an integration of the Alice[4] chatbot engine, which can be used as a fallback, whenever none of the query tools can provide an answer to some user input. For example, it can handle aggressive utterances, such as "You are stupid."

T-RSS: The RSS Tool provides MBEAgents with access to the RSS Manager. Agents can subscribe to RSS feeds and are then informed of newly published articles.

T-TextRelevance: This Tool can rate the relevance of a given piece of text for the user, using a simple keyword-based approach, based on the agent profile collected in user-agent interactions.

T-QuestionAnalysis: A Tool that answers questions about music-related gossip data extracted from web pages (for example personal relationships of artists).

7.4.2 MBE Mind Implementations

The Mind component performs action selection, based on the current input from the environment and the agent's internal state. It can also make use of supporting services, for example the Agent Modeling Server. The following Mind components have been developed in the RASCALLI project:

[4] `http://www.alicebot.org/downloads/programs.html`

Simple Mind: A simple rule-based mechanism for action selection with minimal internal state. The Simple Mind extracts relevant information from the input data and passes the information on to the appropriate effector tool.

DUAL Mind: This Mind implementation incorporates the DUAL/AMBR ([Kok94], [KPss]) cognitive architecture for action selection.[5]

Adaptive Mind: A Mind implementation that performs action selection based on similarities of the current input with experiences from the past, where it received positive or negative feedback from the user for choosing a specific action.[6]

7.5 MBE Agent Definition Layer

Table 7.1 on the following page shows the agent definitions currently implemented for the MBE agent architecture. They are:

Simple Music Companion: An agent definition comprising the Simple Mind and all of the available Tools. It was originally implemented as a proof of concept for the platform and the MBE architecture, but is now also used as a base-line for evaluating other agent definitions.

Adaptive Music Companion: The Adaptive Music Companion is similar to the Simple Music Companion, but uses the Adaptive Mind for action selection.

DUAL Music Companion: This agent definition is based on the DUAL Mind, employing only a subset of the available Tools.

The Agent Factories for these agent definitions are implemented as simple Java classes, making use of iPOJO (see section 5.1.3 on page 38) to select the appropriate Tool components.

7.6 Summary

The Agent Layer is the application layer of the RASCALLI platform, where the actual agents are implemented. In this chapter, we have outlined the

[5]The DUAL Mind has been implemented by project partners NBU and Ontotext

[6]The Adaptive Mind has been implemented by project partner OFAI, based on the Simple Mind.

		Simple Music Companion	**Adaptive Music Companion**	**DUAL Music Companion**
Mind	Simple Mind	x		
	Adaptive Mind		x	
	DUAL Mind			x
Tools	T-IP4Simple	x		
	T-IP4Adaptive		x	
	T-IP4DUAL			x
	T-MMG	x	x	x
	T-QA	x	x	x
	T-TextRelevance	x	x	
	T-RSS	x	x	
	T-NALQI	x	x	
	T-QuestionAnalysis	x	x	
	T-ChatBot	x	x	

Table 7.1: MBE Agent Definitions

implementation of the Mind-Body-Environment (MBE) agent architecture that is currently the basis for the RASCALLI agents.

On a conceptual level, these agents consist of a mind, which makes use of a set of sensor and effector tools (body) to interact with the environment. We have described the core components and roles implemented on the Agent Architecture Layer, as well as the Tool and Mind components implemented on the Agent Component Layer, and how they have been combined to create three agent definitions, the Simple, DUAL and Adaptive Music Companions.

We have also briefly described the 3D client test agents, which serve as a proof-of-concept for the platform's ability to support multiple agent architectures.

Chapter 8

The Platform at Work

In this final chapter, we briefly describe the entire RASCALLI system, as it is currently implemented, from the user's point of view. This includes a short introduction to the user interfaces and the actual functions the agents can perform, as well as pointers to the currently deployed system components. This chapter is meant to provide evidence that the RASCALLI platform has actually been used to implement a running system and to give a more complete picture of the RASCALLI agents.

8.1 RASCALLI User Interfaces

The RASCALLI system currently comprises three major user interfaces, a web interface, a 3D client and a Jabber instant messaging integration, as well as two web-based interfaces for exploration of the music domain.

8.1.1 Getting Started

In order to get started, the user must

- download and install the 3D client,
- install a Jabber client and create a Jabber account, and
- register on the RASCALLI Web Interface.

Detailed instructions for these steps can be found at `http://www2.ofai.at/rascalli/rascalli-getting-started-guide.html`[1]

[1] As of October 21, 2008.

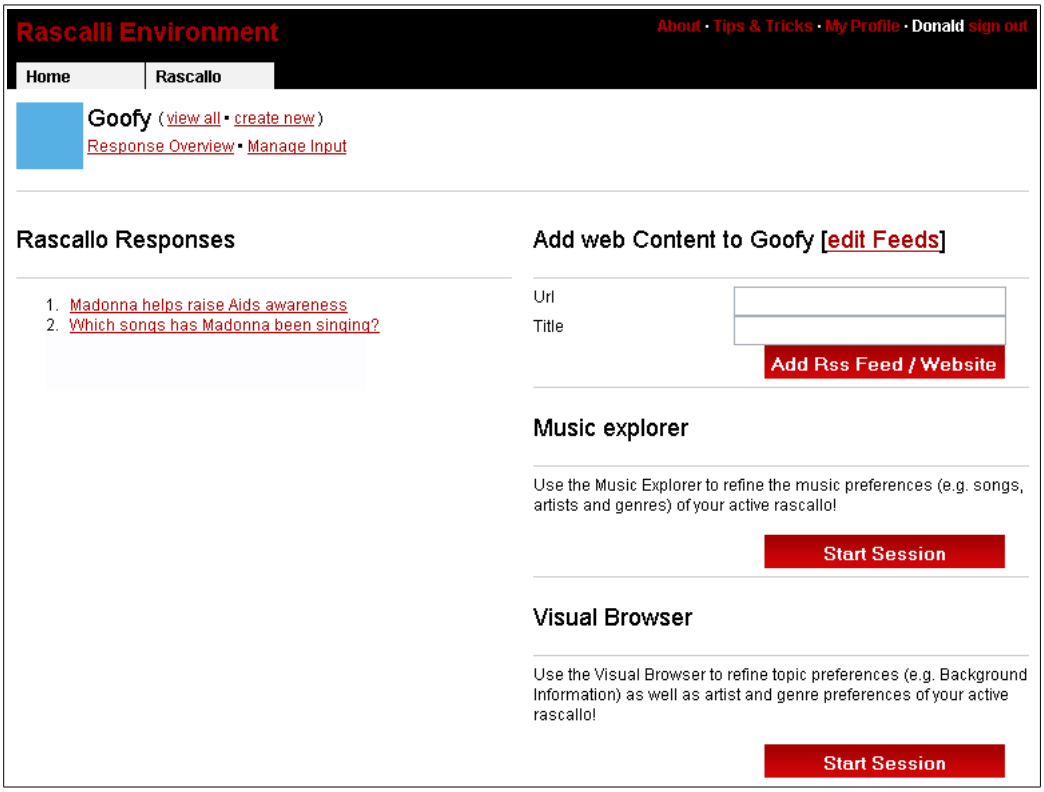

Figure 8.1: RASCALLI Web User Interface

After this initial setup, the user can create agents and start interacting with them via the various user interfaces. The following sections briefly describe the user interfaces and how they can be used to interact with the agents.

8.1.2 Web User Interface

The Web User Interface[2] is the central user interface for RASCALLI. It allows the user to

- create an account (register),
- create agents,
- configure agents (e.g. add or remove RSS feed URLs),

[2]http://intralife.researchstudio.at/rascalli/, as of October 21, 2008.
The Web User Interface has been implemented by project partner SAT.

Figure 8.2: RASCALLI 3D client

- enter one of the browser applications (the Music Explorer and the Visual Browser),
- examine the agent profiles generated from the user interactions, and
- view information posted by the agents (e.g. answers to questions asked in the 3D client or RSS feed items found by the agent).

Figure 8.1 on the preceding page shows the home page of Rascallo Goofy. The left section contains a list of the agent's postings, including an RSS item and the answer to a question. In the right section, the user can add RSS feeds and start browsing sessions in the Music Explorer and Visual Browser.

8.1.3 3D Client

The RASCALLI 3D client[3] is the primary interactive user interface of the RASCALLI system. It is based upon the Nebula open source 3D game engine[4] and displays the agent as a three-dimensional character, sitting in a

[3]The 3D client has been implemented by project partner Radon Labs.

[4]`http://nebuladevice.cubik.org/`

chair in a futuristic room. The character is able to perform bodily gestures and direct its eye gaze, has lip synchronization and uses speech generation, along with written messages for text output.

Figure 8.2 on the previous page is a screen shot of the 3D Client showing the female version of the character (Rascalla). The user can

- send text messages to the agent,
- press the *praise* button,
- press the *scold* button,
- enter one of the browser applications (the Music Explorer and the Visual Browser),
- click on a URL provided as part of an agent's answer to open that URL in the system web browser[5], and
- switch agents.

8.1.4 Jabber

Jabber[6] has been integrated as an alternative interactive user interface mainly for two reasons:

- Instant messaging is a widely accepted mode of interaction on the Internet.
- The minimal resource consumption of a Jabber client, along with its integration with the operating system (e.g. for alerts about incoming messages) increase the availability of the user for agent interaction. This user interface is particularly useful for proactive agents, which can send information to the user as it becomes available (as opposed to reacting to the user's questions).

While the Jabber interface naturally lacks the richness of the 3D client interface, it allows for the same basic mode of interaction: sending text messages. Figure 8.3 on the following page shows a screenshot of a communication with an agent via Jabber.

[5]This is not shown in the screenshot.

[6]Jabber has been integrated in a joint effort with project partner OFAI.

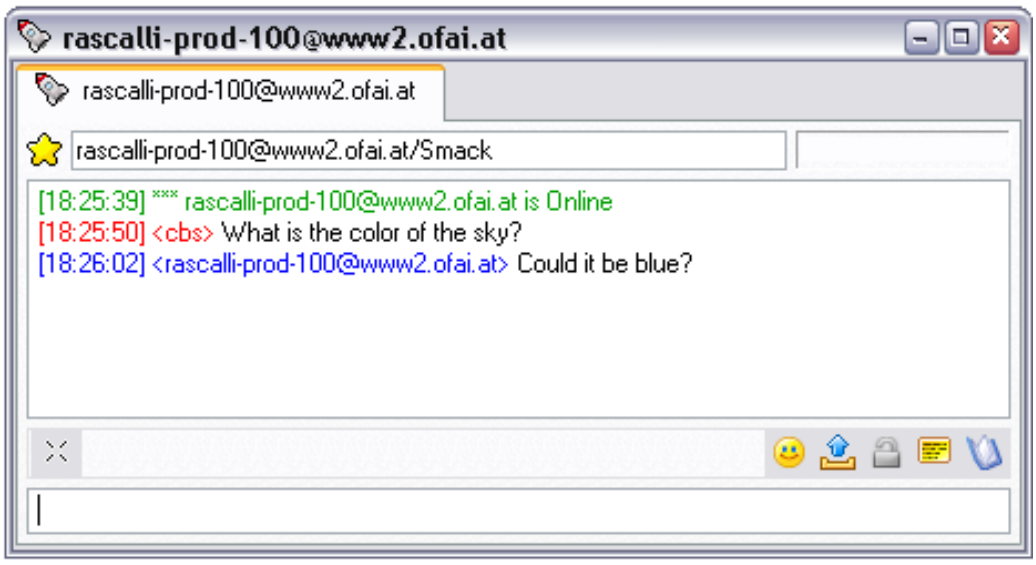

Figure 8.3: Screenshot of a Jabber communication with an agent

8.1.5 Music Explorer

The Music Explorer (MEX)[7] is a web interface for browsing a song database. Users can search for artists, albums and tracks, and listen to those tracks. The most important feature of the MEX is the incorporation of song similarity data, which allows the user to find music similar to the one she is currently listening to. In the screenshot shown in figure 8.4 on the next page, the user is listening to a Madonna song. The other songs in the list (and in the display in the left half) are similar to the current song.

Using this mechanism, the users can start from a specific song they like and explore the music collection based on sound similarity. All user actions, such as listening to or rating a song, are used to generate the agent profile stored in the Agent Modeling Server. This profile is then used by the agent to filter the RSS feeds supplied by the user.

8.1.6 Visual Browser

The Visual Browser[8] (figure 8.5 on page 72) is a web application for browsing music-related gossip data about musicians, including personal relationships, gender, birthday, policital affiliation, and so on. The data has been collected via seed-based relation extraction from unstructured text sources on the In-

[7]The Music Explorer has been implemented by project partner SAT.
[8]The Visual Browser has been implemented by project partner DFKI.

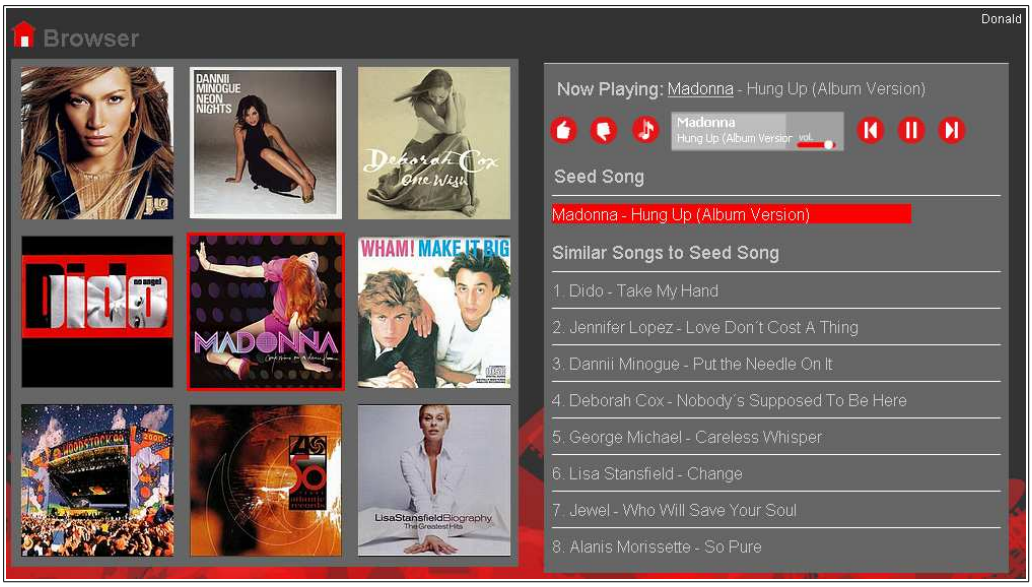

Figure 8.4: Music Explorer

ternet ([XUL07]). Again, the user's actions are used to generate the agent's profile for filtering RSS feeds.

8.2 Available Agents

At the time of writing (October 21, 2008), the DUAL and Adaptive Mind components are still work in progress. While they have already been integrated with the platform and used for the DUAL and Adaptive Music Companion agent definitions, respectively, they have not yet been released for public use. Therefore, only the Simple Music Companion is currently available[9].

The Simple Music Companion can answer music related questions (such as "Who is the husband of Madonna?"), as well as open domain questions (e.g. "What is the color of the sky?"). It can also deal with insults (such as "You are really stupid!").

In addition to these interactive capabilities, the Simple Music Companion monitors the RSS feeds provided by the user. It makes use of the aggregated profile (based on user interactions with the browser interfaces) to perform keyword-based filtering of RSS items. For example, if the profile contains the artist name "Metallica", RSS items about Metallica will be considered

[9]The 3D client test agents are meant for internal use only.

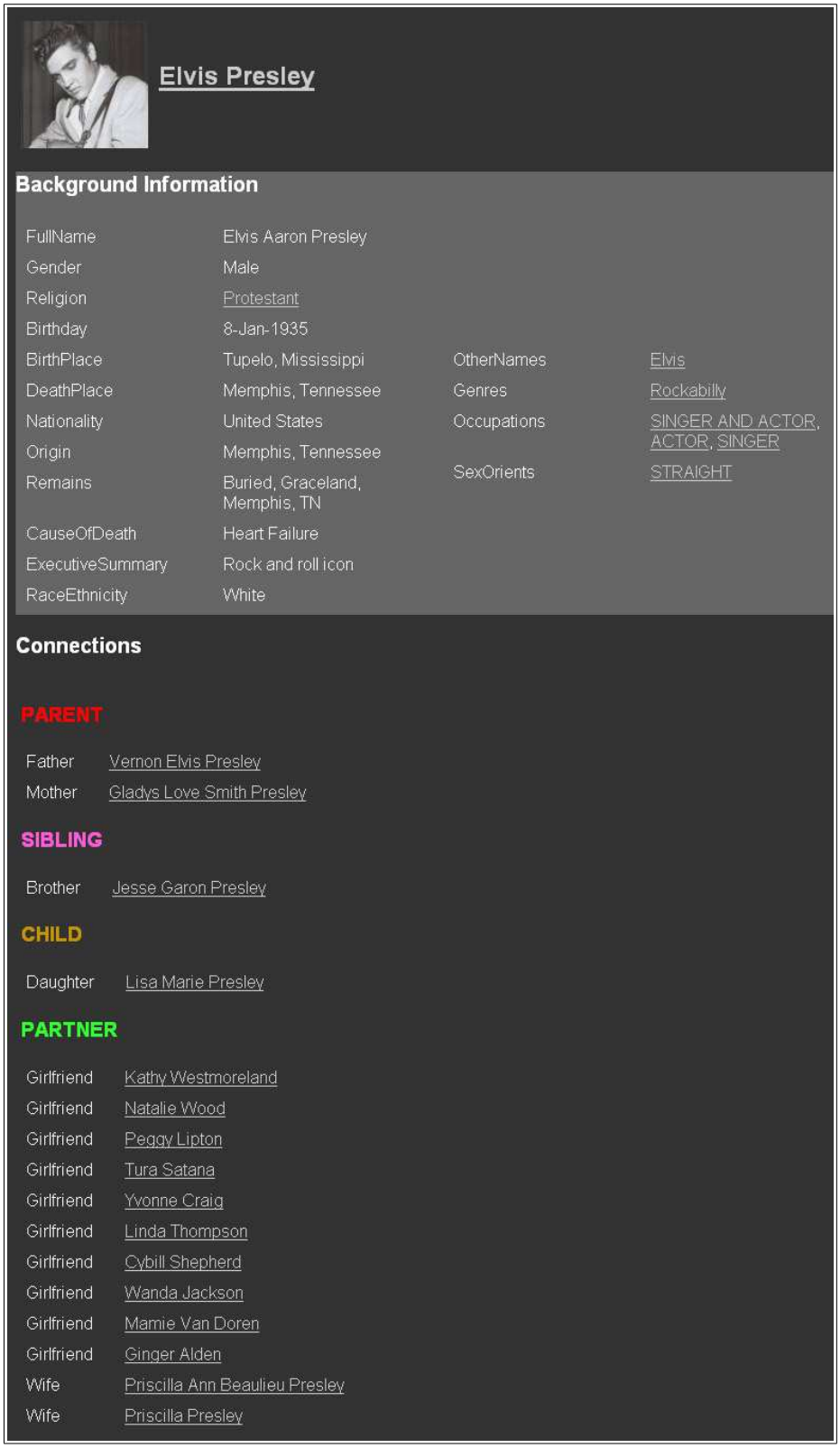

Figure 8.5: Visual Browser

interesting to the user. Interesting items are posted on the agent's homepage and the user is informed via Jabber or the 3D client (depending on the user's availability on either of these communication channels).

8.3 Summary

In this chapter, we presented an overview of the currently implemented RASCALLI system from the user's point of view. We have given a brief introduction to the available user interfaces and the modes of interaction with the Rascalli. Interested readers can follow the steps in the Getting Started Guide to create and interact with an agent.

The RASCALLI platform, as described in the previous chapters, forms the basis of this software system, serves as the integration point of all system components, and executes the actual RASCALLI agents.

Chapter 9

Conclusion and Future Work

In this work, we have presented the RASCALLI platform, an environment for the development and execution of modular software agents. The agents, as well as the entire platform, are implemented as a dynamic component-based system, based on OSGi, which allows for individual components to be added to, updated and removed from the environment at runtime. We have described the three-layered software architecture of the platform, comprising the Infrastructure Layer, the Framework Layer and the Agent Layer. The Agent Layer is again split into four sub-layers, which serve as a conceptual framework for the development of modular agents: Agent Architecture, Agent Component, Agent Definition and Agent Instance Layer.

The primary contribution of this work is the application of component-based software engineering to agent development. The RASCALLI platform supports the execution of *multiple agents*, based on different *agent architectures* and configurations. It is a *shared platform* allowing *multiple users*, developers and agents to use a single instance of the environment. Agents can *communicate* with their users and other agents. The agents, as well as the platform itself, are based on a *component-based architecture*, allowing new agent and platform components to be deployed at runtime.

Promising directions for future work include the integration of existing agent frameworks and cognitive architectures into the RASCALLI platform, as outlined in chapter 3, and the extension of the RASCALLI development environment.

9.1 Extending the Development Environment

The following features are planned to be implemented in order to turn the RASCALLI platform into a fully functional shared development environment:

Web User Interface: A web user interface (WUI) to the OSGi framework will be implemented. This interface will allow multiple developers to access the OSGi container for bundle management at the same time. The WUI will provide the following functionality:

- Managing developer accounts (each developer has a separate account in the platform),
- managing OSGi bundles (bundles can be installed, uninstalled, started, and stopped), and
- support for RASCALLI specific functions, such as managing agent definitions and agents.

Eclipse integration: The Eclipse IDE[1] is built on top of an OSGi container (Equinox). Therefore, it would be rather easy and very convenient to execute a platform instance within Eclipse for development and testing purposes. The basic idea is to implement a simple integration testing framework within Eclipse so that combinations of RASCALLI components can be tested locally before releasing them to the RASCALLI platform.

Maven repository extension: The RASCALLI project makes use of an open-source Maven repository implementation called Artifactory[2]. All RASCALLI components are deployed to this repository and the platform makes use of the repository to install and update the bundles. We plan to implement an extension for Artifactory so that it can be used as an OSGi bundle repository[3].

9.2 Implementing BOD Agents in the RASCALLI Platform

In chapter 3, we have presented a number of agent frameworks and have also pointed out several possible future directions for integrating aspects of those frameworks with the RASCALLI platform. In this section, we outline the

[1] `http://www.eclipse.org/`

[2] `http://www.jfrog.org/sites/artifactory/latest/`

[3] An OBR, also known as Oscar Bundle Repository, can be accessed from within an OSGi container via a special Bundle Repository bundle, which provides an OSGi service for dynamically deploying repository bundles and the transitive closure of their deployment dependencies into an executing OSGi framework (see `http://oscar-osgi.sourceforge.net/`)

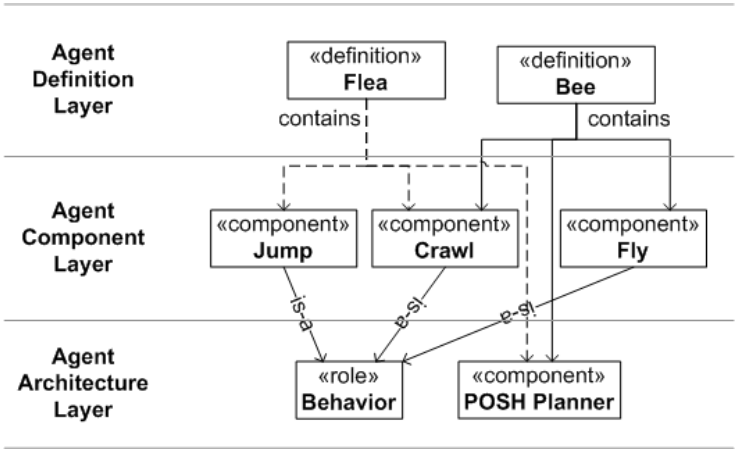

Figure 9.1: BOD Agent Architecture

possible implementation of BOD (see section 3.3) agents within the platform. We have chosen BOD for this section, because the integration would be technically easy and the iterative development process of BOD seems to be a perfect match for the RASCALLI platform.

The basis for BOD agents is POSH action selection. A Java[4] and a Jython[5] implementation of POSH action selection exist and could be integrated as components on the Framework Layer. Building on this central component, we can imagine two viable routes to implement RASCALLI agents based on BOD:

Implement a POSH Mind: Create a Mind implementation for the MBE agent architecture that uses the POSH component for action selection. This would require a mapping of POSH primitives (acts and senses) to MBE sensor and effector Tools.

Implement a new agent architecture: A new agent architecture could be implemented based on BOD. Figure 9.1 shows the components of such an agent architecture. The Agent Architecture Layer defines the role of a *Behavior*. The Behaviors are then implemented on the Agent Component Layer and can be combined into different agent definitions. The POSH planner component implemented on the Framework Layer would use these Behavior components to execute POSH plans.

[4]YAPOSH (Yet Another POSH), `http://technohippy.net/`
[5]jyPOSH, `http://www.cs.bath.ac.uk/~jjb/web/pyposh.html`

List of Figures

List of Tables

Bibliography

[And93] J.R. Anderson. Rules of the mind, 1993.

[AOS08] 9th international workshop on agent oriented software engineering – introduction, 2008. Retrieved Oktober 6, 2008 from `http://grasia.fdi.ucm.es/aose08/`.

[BPR01] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE: a FIPA2000 compliant agent development environment. In *Proceedings of the Fifth International Conference on Autonomous Agents (AGENTS)*, pages 216–217, New York, NY, USA, 2001. ACM.

[Bro86] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, Mar 1986.

[Bry01] Joanna J. Bryson. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS, Cambridge, MA, June 2001. AI Technical Report 2001-003.

[Bry03] Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. In R. Kowalszyk, Jörg P. Müller, H. Tianfield, and R. Unland, editors, *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, pages 61–76. Springer, Berlin, 2003.

[Cle96] Paul C. Clements. From subroutines to subsystems: Component-based software development. In Alan W. Brown, editor, *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, pages 3–6. IEEE Computer Society Press, 1996.

[dSdM08] Paulo Salem da Silva and Ana C. V. de Melo. Reusing models in multi-agent simulation with software components. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1137–1144, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[EHL07] C. Escoffier, R.S. Hall, and P. Lalanda. iPOJO: an extensible service-oriented component framework. In *IEEE International Conference on Services Computing (SCC)*, pages 474–481, Salt Lake City, UT, July 2007.

[Elh99] Michael Elhadad. Event models, 1999. Retrieved September 11, 2008 from `http://www.cs.bgu.ac.il/~elhadad/se/events.html`.

[GL08] Alessandro Garcia and Carlos Lucena. Taming heterogeneous agent architectures. *Commun. ACM*, 51(5):75–81, 2008.

[JSR] JSR 277: Java module system. Retrieved Oktober 3, 2008 from `http://jcp.org/en/jsr/detail?id=277`.

[Kok94] B. Kokinov. A hybrid model of reasoning by analogy. In K.J. Holyoak and J.A. Barnden, editors, *Analogical Connections*, volume 2 of *Advances in Connectionist and Neural Computation Theory*, pages 247–320. Ablex, 1994.

[KPss] B.N. Kokinov and A.A. Petrov. Integration of memory and reasoning in analogy-making: the AMBR model. In D. Gentner, K. Holyoak, and B. Kokinov, editors, *Analogy: Perspectives from Cognitive Science*. MIT Press, in press.

[McI68] Doug McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering*, pages 88–98, Garmisch Pattenkirchen, Germany, 1968. NATO Science Commitee.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[MMM+06] Pragnesh Jay Modi, Spiros Mancoridis, William M. Mongan, William Regli, and Israel Mayk. Towards a reference model for agent-based systems. In *AAMAS '06: Proceedings of the fifth*

international joint conference on Autonomous agents and multiagent systems, pages 1475–1482, New York, NY, USA, 2006. ACM.

[OSG08a] Benefits of using OSGi, 2008. Retrieved September 12, 2008 from `http://www.osgi.org/About/WhyOSGi`.

[OSG08b] The OSGi architecture, 2008. Retrieved September 12, 2008 from `http://www.osgi.org/About/WhatIsOSGi`.

[PC07] Giovanni Pezzulo and Gianguglielmo Calvi. Designing modular architectures in the framework AKIRA. *Multiagent Grid Syst.*, 3(1):65–86, 2007.

[PS96] Cuno Pfister and Clemens Szyperski. Why objects are not enough. In *Proceedings of the First International Component Users Conference (CUC)*, Munich, Germany, July 15–19 1996. SIGS Publishers.

[RAS05] Project RASCALLI, Annex I – "Description of Work", November 2005.

[SA05] M. Skowron and K. Araki. Effectiveness of combined features for machine learning based question classification. *Special Issue of the Journal of the Natural Language Processing Society Japan on Question Answering and Automatic Summarization*, 12(6):63–83, 2005.

[SH04] Jean-Guy Schneider and Jun Han. Components – the past, the present, and the future. In Clemens Szyperski, Wolfgang Weck, and Jan Bosch, editors, *Proceedings of Ninth International Workshop on Component-Oriented Programming (WCOP)*, Oslo, Norway, June 2004.

[Sko05] M. Skowron. *A Web Based Approach to Factoid and Commonsense Knowledge Retrieval*. PhD thesis, Hokkaido University, Sapporo, Japan, 2005.

[Syc98] Katia P. Sycara. Multiagent systems. *AI Magazine*, 10(2):79–93, 1998.

[V+07] Hannes Vilhjálmsson et al. The behavior markup language: Recent developments and challenges. In *Proceedings of the 7th International Conference on Intelligent Virtual Agents (IVA)*, vol-

ume 4722/2007 of *Lecture Notes in Computer Science*, pages 99–111, Paris, France, September 2007. Springer Berlin/Heidelberg.

[XUL07] Feiyu Xu, Hans Uszkoreit, and Hong Li. A seed-driven bottom-up machine learning framework for extracting relations of various complexity. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 584–591, Prague, Czech Republic, June 2007. Association for Computational Linguistics.