
UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES
DEPARTMENT OF
MATHEMATICS AND INFORMATICS


Dejan Mitrović

# Intelligent Multiagent Systems based on Distributed Non-Axiomatic Reasoning

– Doctoral Dissertation –

Advisor:
Prof. dr Mirjana Ivanović

Novi Sad, 2015

# Abstract

The agent technology represents one of the most consistent approaches to distributed artificial intelligence. Agents are characterized by autonomous, reactive, pro-active, and social behavior. In addition, more complex, intelligent agents are often defined in terms of human-like mental attitudes, such as beliefs, desires, and intentions.

This thesis deals with software agents and multiagent systems in several ways. First, it defines a new reasoning architecture for intelligent agents called *Distributed Non-Axiomatic Reasoning System* (DNARS). Instead of the popular Belief-Intention-Desire model, it uses Non-Axiomatic Logic, a formalism developed for the domain of artificial general intelligence. DNARS is highly-scalable, capable of answering questions and deriving new knowledge over large knowledge bases, while, at the same time, concurrently serving large numbers of external clients.

Secondly, the thesis proposes a novel agent runtime environment named *Siebog*. Based on the modern web and enterprise standards, Siebog tries to reduce the gap between the agent technology and industrial applications. Like DNARS, Siebog is a distributed system. Its server side runs on computer clusters and provides advanced functionalities, such as automatic agent load-balancing and fault-tolerance. The client side, on the other hand, runs inside web browsers, and supports a wide variety of hardware and software platforms.

Finally, Siebog depends on DNARS for deploying agents with unique reasoning capabilities.

# Sažetak

Agentska tehnologija predstavlja dosledan pristup razvoju distribuirane veštačke inteligencije. Ono što agente izdvaja od ostalih pristupa su autonomno, reaktivino, pro-aktivno, i socijalno ponašanje. Pored toga, kompleksniji, inteligentni agenti se često definišu koristeći ljudske mentalne konstrukcije, kao što su verovanja, želje i namere.

Disertacija se bavi softverskim agentima i multiagentskim sistemima sa nekoliko aspekata. Prvo, definisana je nova arhitektura za rasuđivanje sa primenom u razvoju inteligentnih agenata, nazvana *Distribuirani sistem za ne-aksiomatsko rasuđivanje* (eng. *Distributed Non-Axiomatic Reasoning System*) (DNARS). Umesto popularnog *BDI* modela za razvoj inteligentnih agenata (eng. *Belief-Desire-Intention*), arhitektura se zasniva na tzv. *Ne-aksiomatskoj logici*, formalizmu razvijenom u domenu veštačke opšte inteligencije. DNARS je skalabilan softverski sistem, sposoban da odgovara na pitanja i da izvodi nove zaključke na osnovu veoma velikih baza znanja, služeći pri tome veliki broj klijenata.

Zatim, u disertaciji je predložena nova multiagentska platforma nazvana *Siebog*. Siebog je zasnovan na modernim standardima za razvoj veb aplikacija, čime pokušava da smanji razliku između multiagentskih sistema i sistema koji se koriste u industriji. Kao DNARS, i Siebog je distribuiran sistem. Na serverskoj strani, Siebog se izvršava na računarskim klasterima, pružajući napredne funkcionalnosti, poput automatske distribucije agenata i otporonosti na greške. Sa klijentske strane, Siebog se izvršava u veb pretraživačima i podržava široku lepezu hardverskih i softverskih platformi.

Konačno, Siebog se oslanja na DNARS za ravoj agenata sa jedinstvenim sposobnostima za rasuđivanje.

# Contents

# List of Figures

# List of Tables

# Preface

*Artificial intelligence* (AI) can be defined as "the science and engineering of making [...] intelligent computer programs" (McCarthy, 2007). Although fairly simple, this definition poses a rather difficult question: what does it mean for a software system to be intelligent?

One of the earliest attempts at describing intelligent software was the famous *Turing test* (Turing, 1950). In this test, a human user is seated in front of two chat windows. One of its chat buddies is a human, and the other one is a software system. If the user cannot distinguish which one is which, then the software is said to be intelligent. Although relatively simple to conduct, over time the test has received many criticisms. For example, it unnecessarily limits the machine's capabilities, forcing it to imitate human behavior, when it could do much more.

This is just one experiment that shows how difficult it is to recognize *intelligence*, or *intelligent behavior* in artificial systems. Nonetheless, a thesis that deals with intelligent machines requires a working definition of intelligence. The definition that best describes the end-goal of the ongoing research presented in this thesis is as follows:

> "Intelligence is the capacity of a system to adapt to its environment while operating with insufficient knowledge and resources." (Wang, 2007, p. 33)

AI has generally been divided into two main branches: *weak* and *strong* (Sharkey and Ziemke, 2001). According to the weak AI, a software system can only imitate human cognition; the best it can do is manipulate symbols, without really understanding them (e.g. Preston and Bishop, 2002).

The strong AI, also known as *Artificial General Intelligence* (AGI), has the long-term goal of developing a software system that can really act like the human mind, that attaches meanings to terms, and has internal desires and beliefs. These *reasoning systems* often consist of a *logical* and a *control* part (Wang, 2006). The logical part includes a formal grammar, a semantic theory, and a set of inference rules for manipulating the symbols; the control part incorporates different types of memories, and a mechanism for resource management.

Both branches of AI have their supporters and opponents, as well as strengths and weaknesses. For example, weak AI is divided into number of techniques, each dedicated to solving a small set of problems, and further divided into incompati-

ble sub-techniques (McCorduck, 2004), with often limiting results. On the other hand, the AGI's end-goal is still pretty much in the realms of science fiction.

## Intelligent agents

Although there is no generally agreed-upon definition of *software agents* (or, simply, *agents*), they can be described as *autonomous* software entity, with various degrees of *intelligence*, capable of exhibiting both *reactive* and *pro-active* behavior (Wooldridge, 1999). An agent may possess many additional characteristics, such as mobility, but the former are often considered to be the defining properties of complex agents.

According to the definition of AI given earlier, an intelligent agent would need to express adaptable and flexible behavior. As discussed in more details in Chapter 1 and by e.g. (Franklin, 2007), intelligent agents can be classified as AGI systems.

An agent rarely exists on its own. Instead, it is a member of an *agent society*. A software system that has a society of agents at its core is referred to as a *multiagent system*. More formally, a multiagent system represents a software system in which a group of agents interacts with each other and the environment in order to solve the problem at hand (Bădică et al., 2011). This interaction can take many different forms, from cooperation, action coordination, and knowledge sharing, to negotiation and mutual competition of self-interested agents. In any case, the social aspect is what distinguishes the agent technology from other artificial intelligence approaches.

## Motivation

The most widely-used approach for designing intelligent agents is the so-called *Belief-Desire-Intention* (BDI) architecture (Rao and Georgeff, 1995). It models the agent in terms of its *beliefs* – statements about the world that might (or might not) be true, *desires* – states of affairs the agent would like to achieve, and *intentions* – desires that the agent is committed to achieving.

Over the years, a number of formalisms for BDI agents have been developed (e.g Cohen and Levesque, 1990; Dunin-Keplicz and Verbrugge, 2010; Guerra-Hernandez et al., 2009; Singh et al., 1999; van der Hoek and Wooldridge, 2013). Although providing strong mathematical backgrounds for developing intelligent agents, these formalisms generally do not consider *practical* applications. That is, they do not take into an account various practical issues, such that the agent might not have enough resources, in terms of time and space, to solve the problem (Wang, 2013).

There are also other practical issues with the BDI model. For example, as noted, the belief is a statement about the world that might or might not be true. However, concrete realizations of the BDI model do not offer the way of expressing the *degree* to which the belief is true; it is often left to the agent

developer to somehow deal with the belief validity. Other critiques of the BDI model include the lack of goal representation, support for learning, planning, and social behavior, etc., although several extensions have been proposed to resolve some of these issues (e.g. Dunin-Keplicz and Verbrugge, 2010; Jarvis et al., 2010; Meneguzzi et al., 2007).

Finally, there is a strong suggestion that using only the three notions of beliefs, desires, and intentions might not be sufficient for modeling human behavior. This shortcoming becomes apparent when the BDI approach is compared to other reasoning and cognitive architecture. As shown in Chapter 6, the AGI research community has generally abandoned the BDI model for constructing "thinking machines," and has instead focused on other, or additional, aspects of human reasoning.

Along with the development of the agent technology, *the web* has recently been transformed into an important software platform. It has gradually evolved into an environment capable of providing functionalities previously available only in desktop applications. The client-side enhancements of the web technology have brought significant benefits to both software developers and end-users. Software developers benefit from the cross-platform support as the same code can be re-used in many different environments. End-users, on the other hand, are given the access to online applications in a variety of ways, without the significant loss of functionalities.

These client-side improvements have been supported by corresponding server-side technologies. This means that, on the server, the focus has been on assuring the *high-availability* of deployed applications, which is concerned with fault-tolerance, scalability, and constant, uninterrupted delivery of services, regardless of software or hardware failures.

In general, the agent research community has followed these ongoing trends of moving the software systems from traditional desktop to web environments. However, the full potential of web and enterprise technologies has yet to be harnessed by a multiagent middleware. For example, most existing middlewares still rely on *Java Applets* for client-side code execution – the approach long surpassed by more advanced technologies.

Therefore, the work presented in this thesis is motivated by two major factors: the shortcomings of the BDI agent architecture, and the lack of support for modern web and enterprise technologies in existing multiagent middlewares.

## Main contributions of the thesis

The main contributions can be briefly summarized as follows.

First, a new architecture for developing intelligent agents is proposed. It abandons the BDI model and its formalisms, and instead uses the so-called *Non-Axiomatic Logic* (NAL), an AGI formalism designed for practical realizations of systems that work under the "assumption of insufficient knowledge and resources" (Wang, 2013). The main novelty of the proposed architecture is in the layered and distributed organization of its backend knowledge base. It is

designed with scalability and fault-tolerance in mind, enabling agents to reason over very large knowledge bases.
Two concrete realizations of the proposed architecture are discussed as well.

Secondly, the thesis proposes an architecture of a unique multiagent middleware. The new middleware is designed to fulfill the functional requirements imposed by modern enterprise and web applications. Many multiagent middlewares do already exist; however, none of them provides all the benefits included in the newly proposed system, including, for example, true platform-independence on the client side, and the support for clustered environments on the server side.

Finally, the thesis presents the very first multiagent system comprised of intelligent agents that rely on distributed non-axiomatic reasoning in order to solve concrete practical problems.

## Thesis organization

The thesis is organized into 3 parts. Part I lays the necessary foundations. Chapter 1 provides a detailed overview of the agent technology. It sets the proper definitions of agents and agent architectures, describes cooperation in agent societies, and discusses most widely-used multiagent frameworks and agent-oriented programming languages. Chapter 2 presents the basic principles of the Non-Axiomatic Logic (NAL), including its syntax and semantics, as well as the inference rules available in the first four layers of the formalism.

Part II presents the main contributions of the thesis. Chapter 3 describes an architecture of the new *Distributed Non-Axiomatic Reasoning System* (DNARS), built using the concepts of NAL and modern scalable software development. It discusses how the system's backend is organized in order to support large-scale knowledge bases of NAL statements, while, at the same time, providing real-time services to large numbers of concurrent users. Two sets of algorithms that realize NAL inference rules in a efficient manner are presented as well.

Chapter 4 starts by introducing our new multiagent middleware, named *Siebog*. As a distributed system, Siebog provides scalability and fault-tolerance on the server, and true platform-independence on the client side. Furthermore, by integrating web and enterprise standards into a unified multiagent framework, Siebog easily achieves cross-platform messaging, agent code sharing, and heterogeneous agent mobility. Finally, the chapter discusses the process of adding support for DNARS-based intelligence and reasoning to Siebog agents.

In Chapter 5, three case studies are presented. The first one shows the benefits of Siebog in practice. The second case study evaluates the run-time efficiency of DNARS, i.e when operating on a large knowledge base and serving high numbers of concurrent users. The final, third case study demonstrates how a multiagent system based on non-axiomatic reasoning can be used to derive new knowledge and solve a concrete problem.

Part III of the thesis is dedicated to the related and future work. Chapter 6 provides a detailed insight into existing related architectures, and highlights the advantages (and disadvantages) of Siebog, both with and without the DNARS-

based reasoning. Finally, Chapter 7 draws the overall conclusions, analyzes the completed work, and proposes future research directions.

# Acknowledgements

First of all, I would like to express my gratitude to my supervisor, dr Mirjana Ivanović, for her valuable help, continuous support, and guidance throughout my PhD studies.

A special recognition is given to dr Zoran Budimac, especially for all the travels abroad that have enabled me to better focus on my research.

My appreciation is extended to dr Milan Vidaković, for sharing his original idea on XJAF, and for helping me to develop it further.

I would also like to thank all the committee members, including dr Costin Bădică, for reading the thesis and providing useful suggestions on how to improve it.

A special thanks goes to dr Pei Wang and the OpenNARS community[1], for helping me to gain a better understanding of NAL and NARS, and for suggesting possible practical applications.

The work on this thesis was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support."

In the end, I would like to thank my friends, family, and colleagues, for supporting me on this journey.

Novi Sad, Dejan Mitrović
May 2015

[1] `https://groups.google.com/forum/#!forum/open-nars`, retrieved on August 12, 2014.

# Part I

# Foundations

# Chapter 1

# An Overview of the Agent Technology

The agent technology includes a wide range of concepts for advanced development of agents and distributed multiagent systems. This chapter provides a general insight into some of the most important concepts: agent architectures, cooperative decision making, platforms and frameworks, agent-oriented programming languages, and mobility. Additional information may be found in e.g. (Bordini et al., 2009; Bădică et al., 2011; de Weerdt and Clement, 2009; Dunin-Keplicz and Verbrugge, 2010; Henderson-Sellers and Giorgini, 2005; Macal and North, 2009; Salamon, 2011; Shoham and Leyton-Brown, 2008; Singh et al., 1999; van der Hoek and Wooldridge, 2012; Vlassis, 2007; Weiss, 1999, 2013).

Many aspects of the agent-based software development are standardized by the *Foundation for Intelligent Physical Agents* (FIPA) (FIPA Home). FIPA is a non-profit governing body for the standardization of agent-related technologies, with the aim of assuring interoperability between different implementations. As an important factor in the design and development of agent-based software, FIPA specifications will be referred to throughout this chapter.

## 1.1 Agents

There are many possible applications of the agent technology, each requiring a different set of functionalities. For example, some problems can be elegantly solved by exploiting mobility (Urra et al., 2010), while for others stationary agents represent better solutions. Because of this, it is often difficult to outline a single, all-encompassing definition of the term *agent*.

The two most thorough definitions are so-called *weak* and *strong notions of agency* (Wooldridge and Jennings, 1995). According to the weak notion, agents are executable (software) entities characterized by *autonomous*, *reactive*, *proactive*, and *social* behavior. *Autonomy* assumes that the agent has a control over its actions and can operate without instructions from an external entity. Agents

are able to *react* to changes in their environment, but also to *take the initiative* and perform actions without external stimuli. Finally, during the pursuit of their goals agents often *communicate* with other entities, such as other agents and human users.

The strong notion extends this definition by including human-like mental attitudes, such as *beliefs*, *desires*, and *intentions*, as well as other advanced concepts: mobility, rationality, benevolence, etc. (Bădică et al., 2011).

Another way of defining agents is by comparing them to objects in object-oriented programming (Wooldridge, 1999). First of all, agents communicate by exchanging messages. Although communication between objects is also defined as *message passing* (Briggs and Werth, 1994), in most actual implementations it is performed by invoking a method on the object reference. In practice, agents are more loosely coupled than objects. Secondly, an agent can decide on its own whether to perform the task or simply ignore the request (e.g. if the request is not in the agent's best interest). When a method is called, the object is expected to execute it, provided that all preconditions are satisfied. Then, agents exhibit pro-active behavior, unlike objects which need to be invoked in order to perform some actions. Finally, an agent has its own thread of control. As discussed by (Wooldridge, 1999), all agent properties can be built into objects. For example, one can easily implement a system of concurrent objects which communicate by sending custom messages to one another, take the initiative, and can choose to ignore receiving requests. However, these are not defining properties of the object-oriented paradigm; one does not expect objects to behave this way.

One of the main topics of this thesis are *intelligent* agents. An intelligent agent is the one exhibiting adaptable and flexible behavior. A more formal definition of intelligent (or, *deliberative*) agents can be specified as follows:

> "We define a deliberative agent or agent architecture to be the one that contains an explicitly represented, symbolic model of the world, and in which decisions (for example about what actions to perform) are made via logical (or at least pseudo-logical) reasoning, based on pattern matching and symbolic manipulation." (Wooldridge and Jennings, 1995, p. 130)

In accordance to the aforementioned definitions of agents, several *architectures* have been proposed. The architecture defines internal organization of an agent: how perceiving the environment triggers actions, for example, or how complex decision-making is performed. The three most widely-used architectures are *reactive*, *Belief-Desire-Intention*, and *hybrid* or *layered* architecture.

### 1.1.1 Reactive agent architecture

Reactive agents operate by continuously adapting their behavior to changes in the environment. By frequently perceiving the environment and producing a small number of actions for each change, they are well-suited for (highly) dynamic environments. Reactive agents generally operate in *reasoning cycles*, with

each cycle consisting of the following operations (Bordini et al., 2007; Salamon, 2011; Wooldridge, 1999):

- The agent observes its environment and creates one or more *percepts*;
- The percepts change the agent's *internal state*; and
- The changes in this internal state trigger one or more actions, which, in turn, affect the environment.

There is a number of possible design approaches for reactive agents, including *finite state machines* (Shalyto et al., 2005), the *subsumption architecture* (Brooks, 1991; Luck et al., 2004; Salamon, 2011), and the *agent network architecture* (Luck et al., 2004; Maes, 1991; Salamon, 2011). An example of a state machine design is shown in Fig. 1.1. It represents a simple football playing agent, with the following possible states (Mitrović et al., 2013b):

- *Idle*: the agent waits for the game to start;
- *Searching ball*: the agent cannot see the ball, and tries to find it;
- *Walking to ball*: the agent sees the ball, but is too far away for the kick;
- *Kicking*: the agent is sufficiently near the ball to perform the kick;
- *Done*: the agent has scored a goal; and
- *Lying down*: the agent has fallen down.

The subsumption architecture is one of the more influential architectures for reactive agents (Brooks, 1991; Luck et al., 2004; Salamon, 2011; Wooldridge, 1999). The agent's behavior is organized into hierarchical layers. Each layer has a set of pre-conditions and a set of actions that are executed if the pre-conditions are met. Because different layers can have their pre-conditions satisfied simultaneously, the execution priority is introduced. Lower layers represent more specific behavior (e.g. *avoid the obstacle*) and have a higher priority than higher layers.

There are several extensions of this original approach. For example, the *dynamic subsumption architecture* allows for layer priority to be changed during the execution (Nakashima and Noda, 1998; Salamon, 2011). Additionally, vertical layering is introduced, where each layer represents a single functionality.

Although seemingly simple and with limited functionalities, reactive agents can be efficiently used to solve complex problems (Nolfi, 2002; Sakellariou, 2014). However, they are domain and environment-dependent, they become increasingly complex as the number of environmental states increases, and finally lack the pro-active component and the possibility of long-term planning and goal-commitment found in deliberative agents (Salamon, 2011).

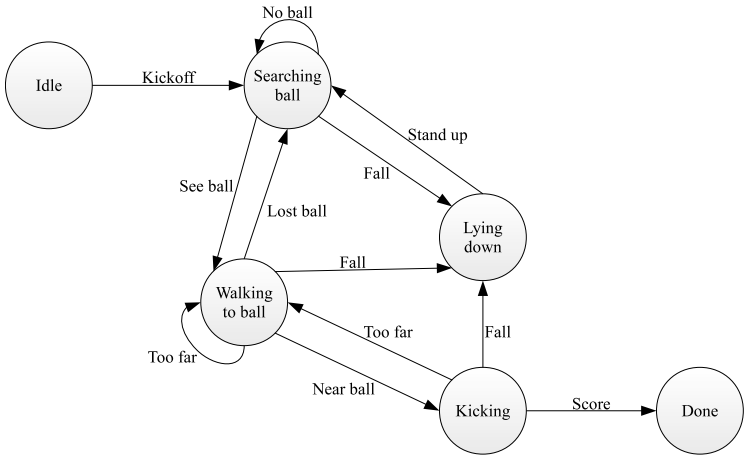



Figure 1.1: Simple reactive football playing agent designed as a finite state machine. Taken from (Mitrović et al., 2013b).
Slika 1.1: Jednostavan reaktivan agent koji igra fudbal, dizajniran kao konačan automat stanja. Preuzeto iz (Mitrović et al., 2013b).

### 1.1.2 Belief-Desire-Intention

The *Belief-Desire-Intention* (BDI) agent architecture was first proposed by (Rao and Georgeff, 1995). Since then, it has had a major influence on the agent technology, becoming the most widely-researched approach for deliberative agent development.

BDI agents are described using human-like mental attitudes, and the notions of *beliefs*, *desires*, and *intentions*. Beliefs represent the agent's knowledge about the environment, which might or might not be true. Desires describe the state of affair the agent will eventually like to achieve. Intentions are desires to which the agent is committed.

The definition of a deliberative agent based on desires and intentions is more of a philosophical nature (Salamon, 2011). In practical systems, such as *Jason* (Bordini and Hubner, 2006), *goals* and *plans* are often used instead of desires and intentions, respectively. The set of goals includes additional restrictions, mainly that there are no conflicting goals. A plan is a concrete sequence of actions designed to achieve a goal.

A general construction of a BDI agent (e.g. Wooldridge, 1999) consists of several components, outlined in Fig. 1.2. Along the beliefs, desires, and intentions, the proposed BDI architecture includes a number of functions:

- *Belief revision*: takes environmental percepts and current beliefs as inputs, and generates a new set of beliefs;

- *Desire generation*: generates agent's desires, based on current beliefs and intentions;

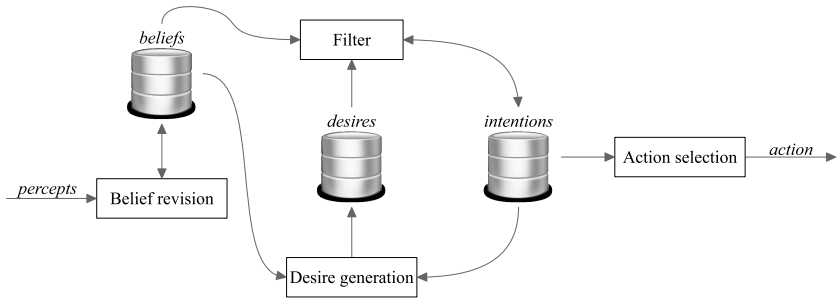



Figure 1.2: General architecture of BDI agents as proposed by (Wooldridge, 1999).
Slika 1.2: Opšta arhitektura BDI agenata, predložena u (Wooldridge, 1999).

- *Filter*: function that generates intentions, based on the agent's current beliefs, desires, and intentions; and
- *Action selection*: chooses an action to be performed on the basis of current intentions.

The filter function represents the agent's deliberation process. Its main purpose is to discard intentions that the agent cannot achieve anymore and to select new intentions, while making sure that there are no conflicts. This process is guided by a pre-defined *commitment strategy*, which determines the agent's level of commitment to a certain intention. Through various experimental evaluations (e.g. Pollack et al., 1994), it has been shown that the optimum commitment strategy depends on the environment: more dynamic environments require more flexible commitment strategy, and vice versa.

One of the strengths of the BDI architecture is the wide range of formalisms that have been developed over time (e.g. Cohen and Levesque, 1990; Dunin-Keplicz and Verbrugge, 2010; Guerra-Hernandez et al., 2009; Singh et al., 1999; van der Hoek and Wooldridge, 2013). In addition, the original design has been extended with other more advanced concepts, such as learning and planning (Phung et al., 2005; Singh et al., 2011), and even emotions (Pereira et al., 2008), making BDI the most feature-rich agent architecture.

### 1.1.3 Hybrid architectures

As indicated by the weak notion of agency, agents need to efficiently combine both reactive and reasoning components. That is, they need to appropriately act to changes in their environment, but also to make decisions and take actions on their own. This is why the so-called *hybrid* (or, *layered*) agent architecture is often used in practical applications. The two most popular hybrid architectures are *TouringMachines* and *InterRRaP* (Luck et al., 2004; Salamon, 2011; Wooldridge, 1999).

Agents based on the TouringMachines model include three main components: *reactive*, *planning*, and *modeling*. The reactive component is at the lower level and should quickly transform percepts into actions. The planning component constructs more complex action plans, while the modeling component maintains models of other agents and the environment, trying to predict future behaviors. Each of the components can directly perceive the environment, and produce actions. In addition, the components themselves are interconnected. Their operation can be managed by a *controller*, which should, for example, choose between conflicting actions proposed by individual components.

Since TouringMachines components can directly communicate with each other and with the environment, this architecture is often described as *horizontally* layered (Luck et al., 2004; Wooldridge, 1999). In InterRRaP, the vertical layering is introduced so that each component/layer can interact only with its direct parent and child layers. The three standard layers of this architecture are, from bottom to top: *behavior-based*, *plan-based*, and *cooperation* layer. The first two correspond to reactive and planning components of TouringMachines, respectively, while the cooperation layer deals with social interactions. Additionally, each layer is associated with its own knowledge of the world.

InterRRaP layers interact in a two-way fashion (Wooldridge, 1999). If a lower layer is incapable of handling the current situation, it will *activate* its parent layer. Similarly, once a higher layer makes a decision on which action(s) to take, it will employ *top-down execution* and forward its decision to the lower layer. By using these decision making approaches, InterRRaP does not require a (complex) control component as TouringMachines.

As noted, InterRRaP includes a layer dedicated to interactions in complex agents societies. In general, this social interaction represents one of the core features of the agent technology, and is discussed in more details in the following section.

## 1.2 Interaction in agent societies

The social ability is one of the defining properties of agents. It has therefore received a great deal of attention from agent researchers and practitioners.

The social interaction can be performed at different abstraction and complexity levels. In the simplest form, agents interact by directly exchanging messages of a pre-defined format. At the more advanced level, agents follow certain norms and *coordinate* their actions, *cooperate* in order to achieve common design goals, resort to *negotiation* for conflict resolution, or, in case of self-interest agents, *compete* against each other (Huhns and Stephens, 1999; Salamon, 2011).

### 1.2.1 Peer-to-peer communication

In order to maintain loose coupling of agents, the following two concepts are generally applied in the design of multiagent systems:

- An agent can reference its peers only through their names/identifiers.

- The communication is asynchronous (non-blocking). It is a well-known fact that asynchronous communication of concurrently executing entities can prevent tricky synchronization issues, such as deadlocks.

These concepts are implemented in and enforced by an agent *middleware*, a runtime environment which provides the infrastructural support for agent execution. It is the job of the agent middleware, for example, to obtain a memory reference to the actual agent and deliver the message to it. Middlewares are described in more details in Section 1.3.

In order to achieve communication-level agent interoperability between different agent vendors (Georgousopoulos et al., 2004), FIPA defines *Agent Communication Language* (ACL) as the standard message format (FIPA Acl). The message is a set of key-value parameters. Standard parameters, many of which are optional, are described in Table 1.1. Custom, user-defined parameters can be included in the message as well, and are prefixed by "X-".

The most important part of the message – its content – can be represented using any language, as long as the receiving agent can understanding it.

Messages based on the FIPA ACL standard include support for interaction protocols. An interaction protocol identifies different roles of agents, and describes the conversation – the order in which different types of messages are expected to be exchanged. One of the most popular protocols is *Contract Net*.

### 1.2.2 The Contract Net protocol

Contract Net is a standard agent interaction protocol (FIPA CNet). It is applicable to scenarios in which an agent has a task that it needs to distribute to other agents. The protocol recognizes two roles: *Initiator* and *Participant*. The general flow of messages between them is as follows:

- Initiator first issues a call for proposals. The call describes the task that needs to be solved, along with any restrictions (e.g. the deadline);
- Interested Participants send their proposals or refuse to participate;
- Initiator analyzes received proposals and selects zero or more Participants to carry out the task. Each bidder will be notified whether its proposal has been accepted or refused;
- Selected Participants execute their tasks. In the end, they inform the Initiator of the result, or about a possible failure.

Therefore, Contract Net can be seen as both a cooperation protocol, when the Initiator engages Participants in order to solve the task, and a competitive protocol, when Participants compete with and try to outbid each other.

Contract Net can be applied to many different real-life scenarios, such as automatic problem resolution in power grids (Kodama et al., 2009). Also, several extensions and improvements of the protocol have been proposed. For example,

Table 1.1: Standard parameters of a message, as defined by the FIPA Agent Communication Language specification.
Tabela 1.1: Standardni parametri poruke, kao što je definisano u FIPA specifikaciji za Agentski Jezik Komunikacije.

| Parameter | Description |
|---|---|
| performative | Type of the message, e.g. request, proposal, agreement, etc. See (FIPA Act) for more information. |
| sender | Message sender. |
| receiver | One or more receivers of the message. |
| reply-to | If specified, identifies the agent to which the reply to this message should be delivered. If omitted, the reply is delivered to the sender. |
| content | Message payload. It contains the actual data the sender is trying to communicate to intended receivers. |
| language | Defines the language in which the content is expressed. |
| encoding | Content encoding. |
| ontology | Content ontology. Together with language and encoding, it helps the receiving agents to understand and interpret the message content. |
| protocol | Identifies the ongoing interaction protocol, if any. |
| conversation-id | Identifies the conversation. It helps the agents to, for example, keep track of conversations with different agents. |
| reply-with | Expression that should be included in the "in-reply-to" parameter that is a reply to this message. |
| in-reply-to | References the "reply-with" parameter in an earlier message to which this message is a reply. |
| reply-by | The deadline by which the agent expects to receive the reply to this message. |

*Competitive Contract Net* introduces support for negotiations regarding contract conclusion and dissolution (Vokřínek et al., 2007).

Contract Net is an example of a one-to-many agent cooperation protocol. When many agents need to share information with one another, an efficient *blackboard* system can be used instead.

### 1.2.3 Blackboards

The blackboard system represents one of the earliest collaboration techniques (Corkill, 2003; Huhns and Stephens, 1999). It is best-suited for cooperative problem solving. A blackboard is a shared repository, analyzed and updated by a set of agent specialists (or, *knowledge sources*). At the beginning, the problem description and any input data are written to the blackboard. Each agent then monitors the changes and makes contributions from its domain of expertise. In general, agents do not communicate with each other directly. Instead, they only see public (and possibly, anonymous) contributions made by other, as well as the overall progress of the problem resolution.

A blackboard consists of a shared memory, communication channels, and an event mechanism (Corkill, 2003). The memory part maintains the progress and history of the problem solving process. It should utilize one of the standard knowledge representation languages (e.g. Schreiber and Raimond), in order to accommodate a diverse group of agents. The communication channel needs to be efficient, and allow for concurrent access and modifications of the memory. Finally, the blackboard solving process is based on events. Agents register themselves to receive notification about events, e.g. when a new piece of information is added.

An integral part of a blackboard system is the control component. It steers the decision-making process, manages priorities between concurrent agents, etc. In more advanced architectures, the control component itself can be designed as a blackboard (Corkill, 2003).

The blackboard system represents an incremental approach to problem resolution, by joining parts contributed by different agents. It can be used to solve complex problems, as long as a diverse group of agents is employed. Therefore, agents in blackboard systems should possess advanced reasoning capabilities.

### 1.2.4 Cooperative learning

The social aspect and cooperation of agents can efficiently be utilized in machine learning. *Cooperative learning* in multiagent systems can be defined as a joint effort of many agents in pursuit of the common goal (Panait and Luke, 2005; Sen and Weiss, 1999). By interacting and sharing their knowledge and experience, agents can learn better and reach the goal faster than isolated systems.

According to the study published by (Panait and Luke, 2005), two main categories of cooperative multiagent learning exist: *team* and *concurrent*. In team learning, a single agent learns for the entire team. The agent observes and tries to improve the team's overall performance, by applying traditional machine

learning techniques. On the other hand, in concurrent learning each member of the team takes part in the learning process: the problem is divided into smaller, more manageable parts that can be handled in parallel.

Each category has its advantages and disadvantages (Panait and Luke, 2005). For example, in team learning the focus is on the performance of the entire team, and not on the needs of individual agents. However, all learning efforts are concentrated in a single agent, which might not be suitable for very complex problems, such as combinatorial optimizations – finding the optimum solution within the large set of possible solutions.

Based on the types of agents, team learning can further be divided into three sub-categories: *homogeneous*, *heterogeneous*, and *hybrid*. In homogeneous learning, all agents have identical capabilities, and so the learned concepts can equally be applied to every member of the team. Heterogeneous learning is concerned with the construction of a diverse team of *specialists*. Although more complicated than homogeneous learning, it can utilize smaller number of agents and provide better results than other machine learning methods (Nitschke et al., 2012; Panait and Luke, 2005). Finally, hybrid team learning combines several distinct groups of homogeneous or heterogeneous agents.

As noted, concurrent learning may be used to efficiently solve more complex problems that team learning, but it possess its own set of challenges (Panait and Luke, 2005; Sen and Weiss, 1999). One notable issue is *credit assignment*, which deals with splitting the reward to individual agents, according to their learning performance. In addition, in order to improve the learning process, the agent should keep models of other agents and adapt to their behavior.

The remainder of this sections discusses two popular cooperative learning approaches used in multiagent systems: *Swarm intelligence* and *Reinforcement learning*.

### 1.2.5 Swarm intelligence

*Swarm intelligence* (SI) belongs to the category of concurrent learning. It is a branch of the so-called *bio-inspired computing*, which tries to mimic the behavior of animals in order to solve complex, usually *NP*-hard problems. SI is primarily focused on social insects and animals, including ants, bees, and birds. The main assumption is that a single agent is relatively simple. But, when acting in a group, their cooperation results in a global intelligent behavior (Blum and Merkle, 2008; Panigrahi et al., 2011). Within the group, there is no centralized monitoring or control. Each agent can only interact with neighboring agents, either directly or through the environment.

The two most popular SI techniques are *particle swarms* and *ant colonies*. Particle swarms are based on the flocking behavior of birds (Blum and Merkle, 2008). Each agent (or *particle*) has a position within the problem search space, it has a velocity, and remembers its earlier position with the optimum value. It can share these information directly with other near-by particles.

Let $p$ be the particle's current position, $v$ its velocity, $p_{opt}$ its optimum position so far, and $p_{glob}$ the optimum position of its neighbors. In each step of the

PSO algorithm the particle updates its information according to the following equations (Blum and Merkle, 2008; Kennedy and Eberhart, 1995):

$$v \leftarrow c_0 v + c_1 R_1 \times (p_{opt} - p) + c_2 R_2 \times (p_{glob} - p) \tag{1.1}$$

$$p \leftarrow p + v \tag{1.2}$$

The three addends of the first equation are interpreted as *momentum*, *cognitive*, and *social* part (Blum and Merkle, 2008). Momentum expresses the particle's tendency to maintain its current velocity. The cognitive part indicates that the particle tends to return to its previous best position. Finally, the social aspect steers the particle towards the best position found by its neighbors. In the equation, $c_0$, $c_1$, and $c_2$ are constants that determine the particle's overall behavior, while $R_1$ and $R_2$ are random values in [0..1].

The ant colony algorithm mimics the behavior of foraging ants (Blum and Merkle, 2008; Dorigo and St uzle, 2004). It uses *pheromone* depositing and detection as the main communication method. Fig. 1.3 demonstrates one classical scenario – choosing the shortest path from the nest to the food source. When an ant reaches the crossroad for the first time (a), it chooses one of the available paths with equal probability. Once it reaches the food source, the ant returns back to the nest, leaving the pheromone trail along the way (shown as a dotted line in (b)). Since now there is a pheromone trail on the upper (shorter) path, there is a higher probability that the third ant will choose this path (c). Over time, more and more pheromone will be deposited on the shorter path. Since the pheromone evaporates, all ants will eventually use only the shorter path.

As noted, SI techniques are best applied to combinatorial optimization problems with large search spaces. One well-known problem in this category is the *Traveling Salesman Problem*, for which there are efficient algorithms based on both ant colonies (Ilie and Badica, 2013) and particle swarms (Yan et al., 2012).

### 1.2.6 Reinforcement learning

*Reinforcement learning* is a well-known machine learning technique based on rewards (Alpaydin, 2004; Busoniu et al., 2008; Sen and Weiss, 1999). Here, the agent is situated in an environment with a set of states $S$, in which it can execute actions from the set $A$. The *state transition function* $f$ is defined as $f : S \times A \rightarrow S$: the agent executes an action $a \in A$, causing the environment to transition from state $s$ to state $p$, $s, p \in S$. For non-deterministic environments, the function is defined as the probability of transitioning to state $p$ when action $a$ is performed in state $s$, $f : S \times A \times S \rightarrow [0, 1]$.

The agent executes actions at distinct time intervals. For each action (or, a *transition step*) the agent receives a scalar reward $r \in R$, which can be negative. The agent's overall behavior is described by a *policy* $\pi$ which maps states to actions, $\pi : S \rightarrow A$. In the non-deterministic case, it is again defined as a probability, $\pi : S \times A \rightarrow [0, 1]$. The goal of the agent is to adopt the policy which maximizes its reward.

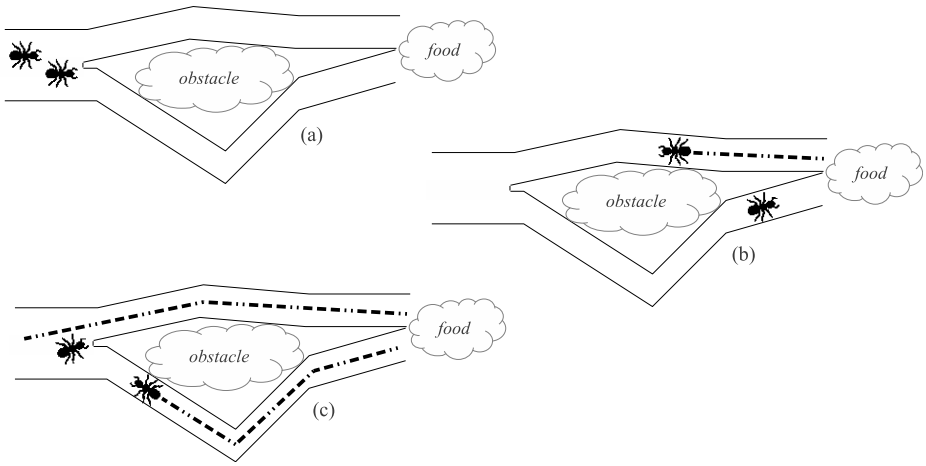



Figure 1.3: The process of selecting the shortest path from the ant nest to the food source. Ants deposit pheromone on their way back from the food source (marked by dotted lines). As the pheromone level increases on the upper path, and decreases (through evaporation) on the lower path, all ants will eventually take the shorter path to the food source.
Slika 1.3: Proces pronalaženja najkraće putanje od gnezda mrava do izvora hrane. Mravi ostavljaju feromonski trag na svom putu od izvora hrane (predstavljen tačkastom linijom). Kako se nivo feromona povećava na gornjoj putanji, a smanjuje (kroz isparavanje) na donjoj, svi mravi će naposletku početi da koriste kraću putanju do izvora hrane.

The most popular variant of Reinforcement learning is the so-called *Q-learning*, which does not require a model of the environment (Alpaydin, 2004; Busoniu et al., 2008). It uses a *Q-function* which determines the reward for each state-action pair for a given policy, $Q^{\pi} : S \times A \rightarrow R$. In practice, these $Q-values$ are calculated iteratively using the greedy policy: selecting the action that returns the maximum reward.

Let the agent follow some greedy policy $\pi$. At each time step $t$ it executes an action $a_t$. This causes the environment to transition from state $s_t$ to $s_{t+1}$, and the agent receives the reward $r_{t+1}$. $Q-values$ are then iteratively updated as follows (Busoniu et al., 2008):

$$Q^{\pi}_{t+1}(s_t, a_t) \leftarrow (1-\alpha)Q^{\pi}_t(s_t, a_t) \ + \alpha(r_{t+1} + \gamma \max_{a \in A} Q^{\pi}_t(s_{t+1}, a)) \qquad (1.3)$$

Here, $\alpha \in (0, 1]$ is the *learning rate*, and it controls how much the new value affects the existing one. The *discount factor* $\gamma \in [0, 1)$ determines the importance of future rewards.

Reinforcement learning is one of the most popular machine learning techniques employed in and by multiagent systems. Some concrete examples of these

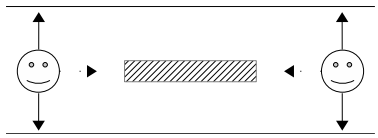


Figure 1.4: Simple action coordination in Reinforcement learning. In addition to choosing the maximum Q-value, agents need to avoid colliding into each other while avoiding the obstacle (adapted from (Busoniu et al., 2008)).
Slika 1.4: Jednostavna koordinacija akcija u Učenju sa podsticajem. Pored odabira najveće Q-vrednosti, agenti paziti da ne dođe do međusobnog sudara pri izbegavanju prepreke (adaptirano iz (Busoniu et al., 2008)).

applications can be found in e.g. (Busoniu et al., 2008; Gabel and Riedmiller, 2007). The technique can be used both in form of a team (single-agent) and a concurrent (multi-agent) cooperative learning. As noted earlier, in the concurrent approach, a special care needs to be taken. For example, Fig. 1.4 shows two agents moving in opposite directions and encountering an obstacle on the road (Busoniu et al., 2008). Each agent can move up or down, and will receive equal reward for either action. However, if both agents move in the same direction, they will collide. Therefore, some form of a strategy or action coordination is required; for example, if two positions result in the same reward, choose the one which is to the right of your moving direction.

## 1.3 Multiagent platforms and frameworks

Agents need infrastructural support in order to perform their tasks. For example, they require an efficient communication infrastructure, which operates regardless of the receiving agents' physical locations. Furthermore, agents need to be able to publish their capabilities to others, or to move to another node in the network. These and other functional requirements have been described and standardized by the FIPA *Agent Management* specification (FIPA Ams).

The three main agent management components are *Agent Management System* (AMS), *Directory Facilitator* (DF), and *Message Transport Service* (MTS). The core functionalities of an AMS are as follows (FIPA Ams):

- Register agents: AMS maintains the list of unique agent identifiers. During the initialization phase, the agent's identifier needs to be registered with the AMS; the agent "officially exists" only after this step.
- Modify the agent description.
- Deregister an agent from the AMS, which makes it unavailable.
- Search for agents registered in the AMS.
- Get the description of the *Agent Platform*.

DF acts as a yellow pages service. Agents can publish (remove, modify) their capabilities and services to the DF, and search the DF in order to learn about other agents. DF does not guarantee the validity of information - description of the agent's service might be outdated, the agent might refuse to execute its service, etc. Finally, an agent might subscribe to the DF and be notified when another agent publishes, removes, or modifies its capabilities.

MTS is the component in charge of transporting messages between agents. A message is comprised of an *envelope* and a *payload* (FIPA Mts). The envelope is a set of key-value parameters, some of which are required (such as *to* and *from*). The payload is context-specific – MTS is concerned with the message envelope only. As discussed in the specification, besides delivering messages to local agents, MTS is expected to enable interaction with remote agents as well, and should provide error handling mechanisms (e.g. when the remote host cannot be reached).

The components are organized inside an *Agent Platform* (AP), also known as the agent *middleware*. Inside an AP, AMS and MTS are mandatory, while DF is optional. There can be any number of DFs in a single AP, in which case the specification assumes that the depth-first search is performed. General organization of an AP is shown in Fig. 1.5. In addition to the core components, AP can include any other component that can support and enhance agents' problem-solving capabilities

As discussed by (Bădică et al., 2011), a large number of agent middlewares has been implemented over the years, but, unfortunately, most of them are not being improved or maintained anymore. The remainder of this section describes several interesting solutions with original design approaches. *Radigost* and *XJAF*, which represent the foundation of this thesis, are described in more details in Chapter 4.

### 1.3.1 Cougaar

*Cognitive Agent Architecture* (Cougaar) is a component based multiagent middleware developed in Java (BBN, 2004; Siracuse et al., 2007). It is designed as a fault-tolerant, scalable distributed architecture. That is, Cougaar and its agents can continue to operate even if a large number of network nodes becomes unavailable. It is, therefore, well-suited for unstable environments.

The platform consists of a number of *nodes* that act as agent hosts. From an external view, a node is just a special type of an agent, executed inside its own Java Virtual Machine.

A Cougaar agent is defined as a set of plug-ins. A plug-in is an independent software component. It does not rely on other plug-ins for its functioning, but can utilize a number of *services*, either readily available in the middleware, or developed by a third-party. Example services include the blackboard, white and yellow pages, logging service, etc.

Blackboard is the main communication interface in Cougaar. It provides asynchronous message exchange, and follows the publish-subscribe model. There is no centralized blackboard, since it would represent the single point of failure.

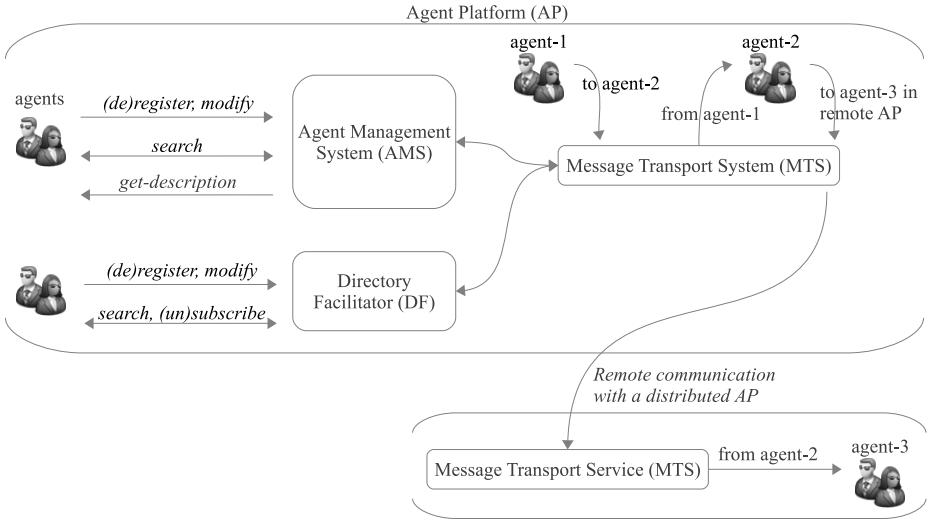



Figure 1.5: Outline of an agent platform comprised of three main components: Agent Management System, Directory Facilitator, and Message Transport Service.
Slika 1.5: Organizacija agentske platforme, koja sadrži tri osnovne komponente: Sistem za upravljanje agentima, Moderatora direktorijuma i Servis za transport poruka.

Instead, each agent has its own blackboard, and can write onto another agent's blackboard, indirectly, through a robust message transport service.

Cougaar offers an advanced persistence sub-system for dealing with hardware and software failures. It can be applied to blackboards, agents, and nodes. The sub-systems preserves internal state of an object on an external storage, and can later restore it, in case of a failure. Two persistence modes are supported: conservative and lazy. The former uses more network traffic and is suitable for highly unstable environments, while the latter is more optimized for stable environments.

Cougaar was developed as an 8-year military project, and it included some advanced solutions for its time. Unfortunately, the system does not appear to be maintained anymore.

### 1.3.2 JADE

*Java Agent DEvelopment Framework* (JADE) is a FIPA-compliant Java-based multiagent framework (Bellifemine et al., 2005, 2007). It is currently the most stable and widely used open-source solution.

JADE *platform* consists of three core components: *Agent Management System*, *Directory Facilitator*, and *Message Transport System*. These components and the platform itself are built in accordance to the FIPA specification de-

scribed earlier. The platform can be executed in form of one or more *containers*. A container is an agent runtime environment. The *main* container is the one that hosts the *AMS* and *DF* components. Other, *peripheral* containers can be distributed across the computer network in order to build distributed systems. Besides distributing agents themselves, this organization provides fault-tolerance through container and agent replication.

Agent communication is achieved through asynchronous message exchange. The message format is based on FIPA ACL. In addition to the simple message passing, JADE supports various interaction protocols, such as *Contract Net*. This allows the development of more complex agents at a higher level of abstraction.

Agents are written primarily in Java, and can be both static and mobile. They are defined in terms of one or more *behaviors*. A behavior represents a task that the agent can perform. In order to simplify the agent-development process, a number of behavior templates are provided, e.g. *cyclic*, *one-shot*, *waker* etc. (Bellifemine et al., 2007). Finally, intelligent agents can be written for JADE using third-party languages and tools, such as *AgentSpeak and Jason*, described in more details in Section 1.4.

JADE has an extensive ecosystem of plug-ins and extensions. For example, a more recent extension named *WADE* (*Workflows and Agents Development Environment*), which operates as a workflow engine on top of JADE. More information about this and other extensions are available at the JADE homepage[1].

### 1.3.3 Magentix

*Magentix* is one of the very few modern multiagent middlewares which does not run in a virtual machine. Instead, it is implemented in C and compiled for Unix-based (i.e. Linux and Mac OS) operating systems (Alberola et al., 2013).

The Magentix platform can be distributed across a number of hosts. A host includes three layers of processes. At the top layer, the *main* process is used to start and control the platform itself as well as other processes. Below it are a number of *services*, including the *Organizational Unit Manager* which adds support for group communication. Finally, agents are executed as child processes of the FIPA AMS service.

Information held by each service is replicated on all hosts. In case of the AMS, for example, this means that every host includes information about agents running in all hosts. This design approach was made for speed purposes, in order to reduce network communication.

As noted, Magentix agents are represented by processes. Each agent consists of three separate threads: for executing the main code, for sending, and for receiving messages. Any number of *mailboxes* can be attached to an agent. Furthermore, for message filtering purposes, a mailbox can be assigned a *conversation identifier*. Subsequently, messages with the given identifier will be routed to the proper mailbox.

[1] `http://jade.tilab.com/`, retrieved on August 12, 2014.

Magentix might lack portability of other (e.g. Java-based) multiagent solutions. However, the main reason for its design is runtime performance: as demonstrated by (Alberola et al., 2013), under heavy loads Magentix performs on the orders of magnitude faster than JADE.

### 1.3.4 SPADE

*Smart Python multi-Agent Development Environment* (SPADE) is a multiagent middleware that utilizes *Extensible Messaging and Presence Protocol* (XMPP) for agent communication (Aranda et al., 2012; Argente et al., 2007). XMPP is an XML-based, open, decentralized, and asynchronous communication protocol[2]. It acts as a layer on top of an existing infrastructure, such as HTTP or WebSocket, and includes many advanced features, such as automatic discovery of participants, postponed delivery to offline clients, etc.

The two main components of SPADE are the multiagent platform, and an agent library. The platform is FIPA-compliant, with the *Message Transport Service* provided in form of an XMPP server. This approach enables SPADE agents to, for example, easily participate in group conversations. The agent library is written in Python, and simplifies the overall agent development process.

Agents are modeled in terms of behaviors, similarly as in JADE. One more advanced behavior available in SPADE is *Finite-State-Machine*, which defines an agent in terms of distinct *states*, as well as *transitions* between these states.

Besides reactive, SPADE also supports deliberative agents based on the BDI model.

## 1.4 Agent-oriented programming languages

Just as there are dedicated, *object-oriented* programming languages, the existence of *agent-oriented* programming languages (AOPL) is crucial for a wider acceptance of the agent technology. AOPLs offer programming constructs that hide the overall complexity of developing agents and allow the developer to focus on solving the problem in question. They represent one of the crucial components of the *agent-oriented programming* paradigm, which views (social) agents as building blocks of software systems (Shoham, 1993).

As with platforms, a relatively large number of AOPLs has been developed. Several popular languages that are still being actively developed and used are described in the remainder of this section. For others, see e.g. (Bordini et al., 2009; Bădică et al., 2011; Dastani et al., 2003; Davies and Edwards, 1994; Shoham, 1993; Thomas, 1994)

### 1.4.1 2APL

*A Practical Agent Programming Language* (2APL) combines declarative and imperative programming styles (Dastani, 2008). It offers several concepts for

[2] `http://xmpp.org/`, retrieved on August 12, 2014.

the programming and execution of agents. These include beliefs, goals, events, actions, plans, and rules. Belief and goals are declarative constructs that describe the agent's mental state. Events carry information about some change in the environment, and are used to trigger plan execution.

Actions describe agent's capabilities, and are divided into six categories. *Belief update* actions modify the belief base, in response to, for example, external events or received messages. *Test* actions query the belief base. *Goal update* actions are used by the agent to drop existing or adopt new goals. *Abstract* actions represent procedure calls, that is, they trigger the execution of plans. *Communication* actions are used for inter-agent communication. Finally, *external* actions are performed by the agent when it needs to affect its environment.

Plans represents the means for achieving the goals. A plan consists of a sequence of actions, with the addition of conditional statements, loops, and non-interleaving operators for building atomic plans. Note that, due to the existence of abstract actions, a plan can, for example, represent a sequence of other plans.

2APL support three types of rules that control the agent's reasoning process (Dastani, 2008). *Planning Goal Rules* create and execute plans in response to some changes in the agent's beliefs and goals. *Procedure Call Rules* create and execute plans in response to environmental changes, received messages, or abstract action. Finally, *Plan Repair Rules* define how a failed plan can be substituted with another plan, as long as some beliefs hold.

As it can be concluded, 2APL provides a rich set of programming constructs for agent development. The language has a strong theoretical basis, and has seen some interesting practical applications (Dastani, 2008).

### 1.4.2 AgentSpeak and Jason

*AgentSpeak* is a declarative, logic-based language first proposed by (Rao, 1996). Although envisioned as an abstract language, AgentSpeak has recently gained more popularity due to the development of a practical interpreter named *Jason* (Bordini and Hubner, 2006; Bordini et al., 2007).

AgentSpeak/Jason agents are defined in terms of beliefs, goals, and plans. Beliefs are expressed as first-order logical formulae. Goals can be either *test* or *achievement*. Test goals simply query the belief base in order to determine whether some facts are true or false. Achievement goals, on the other hand, trigger the execution of plans.

Plans outline the set of actions for achievement goals. They are specified in the form of $triggeringEvent : context \leftarrow body$ (Bordini et al., 2007; Mitrović et al., 2013b). Once there is a change in the agent's mental state (e.g. a new belief has been added), a plan with such triggering event is selected. But, before the plan is executed, the validity of its context is evaluated. This increases the chances of the successful plan execution and is useful in dynamic environments.

Jason is designed to be highly customizable. Besides the possibility of implementing custom actions in Java, different components of the interpreter itself can be replaced. This design approach has undoubtedly contributed to the popularity and practical applicability of Jason, allowing its agents to run on

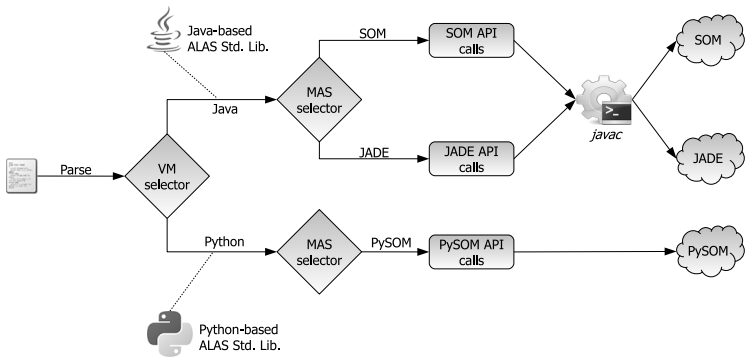



Figure 1.6: *On-the-fly* compilation of an ALAS source code into the executable code of the concrete multiagent platform (taken from (Mitrović et al., 2012a)).
Slika 1.6: *U-letu* prevođenje izvornog koda pisanog u ALAS-u u izvršni kod konkretne multiagentske platforme (preuzeto iz (Mitrović et al., 2012a)).

different software infrastructures, such as JADE (Bordini et al., 2007) and Java EE (Mitrović et al., 2013a).

### 1.4.3 ALAS

Unlike other languages described in this section, *ALAS*[3] is an imperative language, suitable for reactive agent architectures (Mitrović et al., 2011; Mitrović et al., 2012a). One of its main goals is to support agent mobility between platforms that offer different sets of APIs, or that are executed on top of different virtual machines. This goal is achieved by recompiling the agent's source code *on-the-fly*, as it reaches the target platform.

The re-compilation process is shown in details in Fig. 1.6. First, the ALAS source code is transformed into an abstract syntax tree and then fed into *VM selector*, which associates it with the correct standard library. Then, *MAS selector* replaces all platform-related calls to the appropriate API calls offered by the target platform. Finally, the resulting code is forwarded to the native compiler to produce the executable code.

ALAS agents are defined in terms of *services* and *properties* (Mitrović et al., 2011; Mitrović et al., 2012a). A service is a distinct piece of functionality that the agent offers to external entities. Service execution can be requested by sending an appropriate message to the agent. Properties describe the state of an agent. They can be either *persistent* or *transient*. During the migration process, only persistent properties are transferred along with the agent, reducing the overhead of agent mobility.

[3]Developed at the University of Novi Sad, Serbia.

### 1.4.4 GOAL

Mental state of an agent written in GOAL (*Goal-Oriented Agent Language*) consists of *knowledge*, *beliefs*, and *goals* (Hindriks, 2014, 2009). The difference between knowledge and beliefs is that the former is static and cannot be changed at runtime, while the belief base can (and is) continuously updated. All three constructs are written declaratively, using Prolog-like facts and rules.

The language supports different types of goals, including *achievement* and *maintenance* goals (Hindriks, 2014, 2009). In both cases, the goal describes desired environment state. However, in case of achievement goals, the environment is not currently in that desired state, and the agent needs to act in order to reach it. In case of maintenance goals, the agent can refrain itself from acting in order to keep the current state.

GOAL agents follow the *blind commitment strategy* (Hindriks, 2014, 2009; Rao and Georgeff, 1993). This means that the agent will not drop an active goal until it is fully completed (i.e. including all sub-goals).

Actions are mostly user-defined, and are guarded by precondition and post-condition expressions. If its precondition can be derived from the agent's beliefs, the action is said to be *enabled*. The postcondition is generally split into two lists, *add* and *delete*, consisting of, respectively, positive and negative literals. Once the action is executed, literals in the delete list are removed from the belief base, and only then are literals in the add list included.

## 1.5 Mobility

During the mid-1990s, mobility was the defining property of agents (which were often referred to as *mobile agents*). After the initial "hype," it became apparent that mobility does not represent the solution to all problems, and that in practice it often adds more overhead than the theory might suggest (Carzaniga et al., 2007). Nonetheless, mobility is still an interesting research topic, with some important problems yet to be efficiently solved (Medvidovic and Edwards, 2010).

The mobility process consists of several stages (Cabri et al., 2000; Mitrović et al., 2011):

1. Suspend the agent's execution flow;
2. Store its runtime state;
3. Migrate the agent's code and state to the destination machine;
4. Restore the runtime state; and
5. Resume the agent's execution flow.

Depending on how these stages are realized, there exist two types of mobility: *weak* and *strong* (Cabri et al., 2000; Mitrović et al., 2011). With weak mobility, only (some or all of) agent's properties are transfered, and the agent's execution is restored by, for example, sending it a predefined message. With strong

mobility, the entire runtime state, including the execution stack, the program counter, etc. is transferred. Weak mobility is easier to implement, as it does not require any support by the underlying infrastructure (e.g. the virtual machine). Strong mobility is more transparent, but due to various technical difficulties, it is supported only by a small number of usually older systems (e.g. Suri et al., 2000; Tardo and Valente, 1996).

One of the technical difficulties in implementing agent mobility is the lack of interoperability between existing agent platforms (Pinsdorf and Roth, 2002). This lack of interoperability often appears inadvertently, e.g. due to the usage of different virtual machines, by exposing different sets of APIs to agents, etc. Depending on the types of platforms that comprise the network, agent mobility can be classified as follows (Mitrović et al., 2011; Overeinder et al., 2006):

1. *Homogeneous*: all platforms offer the same API and are running on the same virtual machine;

2. *Cross-platform*: offered APIs differ, but platforms run on the same virtual machine;

3. *Agent-regeneration*: virtual machines differ, but each platform offers the same API; and

4. *Heterogeneous*: all platforms offer different sets of APIs and are running on different virtual machines.

Cross-platform mobility can be achieved through software layering, by building an intermediary API layer between agents and platforms (Fortino et al., 2008; Grimstrup et al.). In case of 3., the agent's executable code needs to be regenerated for each virtual machine the agent visits. Obviously, heterogeneous mobility poses the most difficult technical challenges. *Generative migration* (Overeinder et al., 2006), the usage of *model-driven engineering* (Gherbi et al., 2009), and ALAS (Mitrović et al., 2011; Mitrović et al., 2012a) represent possible approaches to achieving full heterogeneous mobility.

## 1.6 Summary

The agent technology includes a wide variety of theoretical concepts and practical solutions for building distributed, autonomous, deliberative systems. In addition to introducing new concepts, such as autonomy and mobility, it has successfully enhanced many existing approaches, such as multiagent reinforcement learning. As a research field, the agent technology is very active, with numerous high-impact journals and international conferences dedicated to the latest breakthroughs in the field.

Unfortunately, agents are yet to be adopted by the mainstream industry. Researchers generally agree that there is no "killer application" for the agent technology – the application that undoubtedly proves its core value. Nonetheless,

over the years the advantages of agents over other approaches have been shown many times.

This chapter has provided an overview of some of the most important research topics in the field of software agents. Besides studying these topics further, interested readers are also encouraged to explore other concepts, including multiagent planning (de Weerdt and Clement, 2009), agent formalisms (Dunin-Keplicz and Verbrugge, 2010; Singh et al., 1999; van der Hoek and Wooldridge, 2012), agent-based modeling and simulation (Macal and North, 2009), etc.

In the next chapter, a formalism named *Non-Axiomatic Logic* is presented. It serves as the basis for a distributed non-axiomatic reasoning system proposed later, in Chapter 3.

# Chapter 2

# Non-Axiomatic Reasoning

*Non-Axiomatic Logic* (NAL) represents a formalism for reasoning systems in the domain of Artificial General Intelligence (AGI) (Wang, 2006, 2013; Wang and Awan, 2011). It includes a symbolic grammar, a set of inference rules, and a semantic theory. However, NAL is different from many other logics used to define reasoning in intelligent systems, in the sense that it is a *term* logic (Smith, 2012; Sommers and Englebretsen, 2000; Wang, 2013). Its sentences are given in the form of *subject-copula-predicate*, where *subject* and *predicate* are terms.

The term *non-axiomatic* in NAL indicates that the logic is constructed around the notion of *insufficient knowledge and resources* (Wang, 2013; Wang and Awan, 2011). In fact, this notion is one of defining characteristics of NAL, and it encompasses several important concepts.

First of all, knowledge is uncertain, and not necessarily consistent. New evidence can be accepted at any time, it can include any content, and can affect the truth of any existing statement. But, this truth is not expected to converge to any value. At the same time, the system usually does not have enough resources, in terms of space and time, to consult its entire knowledge base when solving a problem. It cannot apply the full set of inference rules, nor follow a predefined algorithm. Finally, the problem-solving process is localized, in the sense that only a fraction of statements is used to reach the conclusion.

NAL includes built-in mechanisms for dealing with the aforementioned issues. It can efficiently manage uncertainty and statement inconsistencies, and summarize existing knowledge in order to reduce the sheer amount of statements. Besides providing sound theoretical basis for logical reasoning, NAL is, therefore, highly practical, and can be efficiently realized using the present-day technology.

NAL is organized into 9 layers. Each layer builds on top of the previous one, by introducing new concepts, grammar, and/or inference rules. Briefly, each layer includes the following (Wang, 2013):

- *NAL-1*: Inference rules for *inheritance*.
- *NAL-2*: *Similarity*, *instance*, and *property* copulas.

- *NAL-3*: Compound terms.
- *NAL-4*: Arbitrary relations among terms.
- *NAL-5*: Higher-order statements.
- *NAL-6*: Variables.
- *NAL-7*: The concept of time.
- *NAL-8*: Support for *operations*, i.e. procedural statements.
- *NAL-9*: Self-control and self-monitoring.

The work presented in this thesis is roughly based on layers 1 – 4, which will be discussed in the remainder of this chapter. As shown later in Chapter 5, the use of the first four layers is sufficient for implementing a system that achieves concrete practical results. Realization of the remaining layers, which are required for more advanced reasoning, is planned for future work (Chapter 7). For a more comprehensive theoretical discussions about these and other NAL layers, see e.g. (Wang, 2006, 2013).

## 2.1 Experience and truth-values

As noted, NAL is a term logic that operates on statements in the form of *subject-copula-predicate*. The basic and most common type of a statement is *inheritance*.

**Definition 2.1.** *Inheritance statement* in NAL is a statement in the form of $S \rightarrow P$, where $S$ and $P$ are terms denoting the subject and object, respectively, and $\rightarrow$ denotes the inheritance copula (Wang, 1994, 2006, 2013).

The inheritance statement $S \rightarrow P$ can be read as *S is a type of P*, for example: *cat is a type of animal*. Subject and predicate are *atomic* or *compound* terms[1]. An atomic term is a word consisting of characters from a finite alphabet. Compound terms are build by connecting atomic or compound terms (Section 2.5). By definition, inheritance is transitive and reflexive. Inheritance statements in the form of $S \rightarrow S$ are called *tautologies*, and are often excluded from the system's experience for redundancy reasons (Wang, 2013).

NAL has an *experience-grounded semantic* (Rodriguez and Geldart, 2009; Wang, 2005), which is based on the concepts of *specializations* and *generalizations*.

**Definition 2.2.** In an inheritance statement $S \rightarrow P$, $S$ is said to be the *specialization* of $P$, while $P$ is said to be the *generalization* of $S$ (Wang, 2013).

[1] In higher NAL layers, subject and object themselves can also be statements.

Let $V_K$ be the set of all terms appearing in system's experience $K$. The *extension* $T^E$ and *intension* $T^I$ of a term $T \in V_K$ can be defined using specializations and generalizations as follows (Wang, 2011, 2013):

$$T^E = \{x|(x \in V_K)(x \to T)\} \tag{2.1}$$
$$T^I = \{x|(x \in V_K)(T \to x)\} \tag{2.2}$$

The *evidence* for a term $T$ (or, its *meaning*) consists of both $T^E$ and $T^I$. That is, the meaning of a term is defined through its relations with other terms. The term has a meaning for the system only if it appears in its experience; otherwise, it is meaningless and has no interpretation.

For a statement $S \to P$, *positive* and *negative* evidence, denoted as $E^+$ and $E^-$, respectively, are defined as follows (Wang, 2011, 2013):

$$E^+ = \{S^E \cap P^E\} \cup \{P^I \cap S^I\} \tag{2.3}$$
$$E^- = \{S^E \setminus P^E\} \cup \{P^I \setminus S^I\} \tag{2.4}$$

The amount of positive evidence $w^+$ represents the cardinality of $E^+$, while the amount of negative evidence $w^-$ represents the cardinality of $E^-$. The amount of total evidence $w$ is calculated as $w^+ + w^-$.

An example shown in Table 2.1 outlines the initial experience of a system which consists of four inheritance statements. The first three statements describe the flying bat as a type of animal and mammal. In the last statement the term bat refers to a club, and the statement indicates that baton is a type of bat (e.g. used by law enforcement).

In order to collect evidence for the statement *bat* $\to$ *mammal*, we construct extension and intension sets shown in the second column of Table 2.1. Then, by applying Eq. 2.3 and 2.4, we can determine that *animal* and *bat* represent positive evidence for the given statement, while *baton* represents negative evidence.

Positive and negative evidence is used to determine the *truth-value* of a NAL statement.

**Definition 2.3.** The truth-value of a NAL statement is represented by a pair of real numbers in $[0, 1]$, named *frequency* ($f$) and *confidence* ($c$) (Wang, 1994, 2001b).

Frequency is the ratio of positive and total evidence, while confidence describes how stable this frequency will be when the system gains new evidence (Wang, 1994, 2001b):

$$f = w^+/w \tag{2.5}$$
$$c = w/(w + k) \tag{2.6}$$

Here, $k$ is the *evidential horizon*, a constant used to prevent the system from comparing possibly infinite future to the relatively short past. When a unit

Table 2.1: Example of the initial knowledge base, as well as the extension, intension, and evidence sets for the statement $bat \rightarrow mammal$.
Tabela 2.1: Primer početne baze znanja, proširujućih i sužavajućih skupova, kao i skupa dokaza za rečenicu $bat \rightarrow mammal$.

| Knowledge base | Extensions & intensions | Evidence |
|---|---|---|
| $bat \rightarrow animal$ | $S^E = \{bat, baton\}$ | $E^+ = \{animal, bat\}$ |
| $bat \rightarrow mammal$ | $P^E = \{animal, bat, mammal\}$ | $E^- = \{baton\}$ |
| $mammal \rightarrow animal$ | $S^I = \{animal, bat, mammal\}$ | |
| $baton \rightarrow bat$ | $P^I = \{animal\}$ | |

amount of future evidence is considered (i.e. $k = 1$), the truth-value of the statement $bat \rightarrow mammal$ is $\langle f, c\rangle = \langle 0.67, 0.75\rangle$.

These specifications of evidence and truth-values lay the foundations for formal reasoning in NAL.

**Definition 2.4.** *Inference rules* in NAL are used to derive new knowledge, to provide answers to questions, or to deal with statement inconsistencies, referred to as *forward*, *backward*, and *local*, respectively (Wang, 2013).

Most of the rules take the *syllogistic* form (Smith, 2012). A syllogistic inference rule takes two premises that share a term, and then derives a conclusion consisting of the remaining two terms. Depending on the copulas and positions of the shared term in premises, different inference rules can be applied.

## 2.2 Forward inference rules

As noted, forward inference rules are used to derive new knowledge. The three most important inference rules that operate on inheritance statements are *deduction*, *induction*, and *abduction*, defined respectively as follows (Wang, 2001a, 2013):

$$\{M \rightarrow P\langle f_1, c_1\rangle, S \rightarrow M\langle f_2, c_2\rangle\} \vdash S \rightarrow P\langle f_1 f_2, f_1 f_2 c_1 c_2\rangle \tag{2.7}$$

$$\{M \rightarrow P\langle f_1, c_1\rangle, M \rightarrow S\langle f_2, c_2\rangle\} \vdash S \rightarrow P\langle f_1, \frac{f_2 c_1 c_2}{f_2 c_1 c_2 + k}\rangle \tag{2.8}$$

$$\{P \rightarrow M\langle f_1, c_1\rangle, S \rightarrow M\langle f_2, c_2\rangle\} \vdash S \rightarrow P\langle f_2, \frac{f_1 c_1 c_2}{f_2 c_1 c_2 + k}\rangle \tag{2.9}$$

Additional rules can be defined as different forms of these three rules. For example, *exemplification* is similar to deduction, but derives the conclusion in an opposite direction, while *conversion* is a form of abduction when terms $S$ and $M$ are the same.

Truth-value functions are defined first by treating frequency and confidence as *extended boolean variables*, which take their value in $[0, 1]$. The set of operators for working with extended boolean variables is defined as follows (Wang, 2001a, 2007, 2013):

$$not(x) = 1 - x \tag{2.10}$$
$$and(x_1, x_2, \ldots, x_n) = x_1 \times x_2 \times \cdots \times x_n \tag{2.11}$$
$$or(x_1, x_2, \ldots, x_n) = 1 - ((1 - x_1) \times (1 - x_2) \times \cdots \times (1 - x_n)) \tag{2.12}$$

For each inference rule, relationships between positive and negative evidence of premises and the conclusion are analyzed. Truth-value of the conclusion is then specified as a function of extended boolean operators on frequency and confidence. For example, truth-value function for deduction can be written as $f = and(f_1, f_2)$, $c = and(f_1, f_2, c_1, c_2)$, which results in the expressions shown earlier.

## 2.3 The similarity copula

To increase the expressive power of NAL, a new copula – *similarity* – is introduced.

**Definition 2.5.** Similarity is denoted by $\leftrightarrow$ , and can be defined as a symmetric inheritance: $(S \leftrightarrow P) \Leftrightarrow (S \rightarrow P) \wedge (P \rightarrow S)$ (Wang, 2009, 2013).

Intuitively, the similarity statement $S \leftrightarrow P$ indicates that $S$ and $P$ are identical to each other. By definition, similarity is transitive, reflexive, and symmetric. As with inheritance, similarity statements in the form of $S \leftrightarrow S$ are tautologies, and are usually excluded from the system's experience.

Similarity copula brings three new forward inference rules: *comparison*, *analogy*, and *resemblance* (Wang, 2006, 2009, 2013). Comparison takes two inheritance premises as those in induction and abduction, but derives a similarity statement. Two versions of the rule exist: extensional and intensional, corresponding respectively to induction and abduction (Wang, 2006, 2013):

$$\{M \rightarrow P\langle f_1, c_1\rangle, M \rightarrow S\langle f_2, c_2\rangle\} \vdash S \leftrightarrow P\langle f, c\rangle \tag{2.13}$$
$$\{P \rightarrow M\langle f_1, c_1\rangle, S \rightarrow M\langle f_2, c_2\rangle\} \vdash S \leftrightarrow P\langle f, c\rangle \tag{2.14}$$

For example, from the two premises $tiger \rightarrow cat$ and $lion \rightarrow cat$, intensional comparison derives the conclusion that $lion \leftrightarrow tiger$. The truth-value of the conclusion can be calculated using the following pair of equations:

$$f = f_1 f_2 / (f_1 + f_2 - f_1 f_2) \tag{2.15}$$
$$c = ((f_1 + f_2 - f_1 f_2) c_1 c_2) / ((f_1 + f_2 - f_1 f_2) c_1 c_2 + k) \tag{2.16}$$

Analogy operates as deduction, but with one premise being a similarity statement. Depending on the position of the shared term in premises, there can be four versions of the rule (Wang, 2006, 2009, 2013):

$$\{M \to P\langle f_1, c_1\rangle, S \leftrightarrow M\langle f_2, c_2\rangle\} \vdash S \to P\langle f, c\rangle \tag{2.17}$$
$$\{P \to M\langle f_1, c_1\rangle, S \leftrightarrow M\langle f_2, c_2\rangle\} \vdash P \to S\langle f, c\rangle \tag{2.18}$$
$$\{M \leftrightarrow P\langle f_1, c_1\rangle, S \to M\langle f_2, c_2\rangle\} \vdash S \to P\langle f', c'\rangle \tag{2.19}$$
$$\{M \leftrightarrow P\langle f_1, c_1\rangle, M \to S\langle f_2, c_2\rangle\} \vdash P \to S\langle f', c'\rangle \tag{2.20}$$

For example, from the two premises $cat \to animal$ and $feline \leftrightarrow cat$, the first version of analogy concludes that $feline \to animal$. In case of analogy, truth-values of conclusions are calculated using the following sets of equations:

$$f = f_1 f_2 \tag{2.21}$$
$$c = f_2 c_1 c_2 \tag{2.22}$$
$$f' = f_1 f_2 \tag{2.23}$$
$$c' = f_1 c_1 c_2 \tag{2.24}$$

Finally, resemblance derives a similarity conclusion from two similarities, and is defined as (Wang, 2006, 2013):

$$\{M \leftrightarrow P\langle f_1, c_1\rangle, S \leftrightarrow M\langle f_2, c_2\rangle\} \vdash S \leftrightarrow P\langle f_1 f_2, (f_1 + f_2 - f_1 f_2) c_1 c_2\rangle \tag{2.25}$$

## 2.4 Instance and property copulas

The NAL-2 layer introduces two additional copulas: *instance*, and *property*, denoted as $\circ\!\!\to$ and $\to\!\!\circ$, respectively (Wang, 2006, 2013). However, two new concepts are required to define these new copulas: *extensional* and *intensional* sets.

**Definition 2.6.** If $T$ is a term, the *extensional set* that contains only $T$, denoted as $\{T\}$, is defined as $(\forall x)((x \to \{T\}) \Leftrightarrow (x \leftrightarrow \{T\}))$ (Wang, 2006, 2013).

**Definition 2.7.** If $T$ is a term, the *intensional set* that contains only $T$, denoted as $[T]$, is defined as $(\forall x)(([T] \to x) \Leftrightarrow ([T] \leftrightarrow x))$ (Wang, 2006, 2013).

Intuitively, an extensional set represents the most specialized term – one that no other term can inherit. An intensional set represents the most general term, i.e. one that inherits no other term. For example, statement $\{Lefty\} \to dog$ indicates that $Lefty$ is one specific dog, while the statement $animal \to [alive]$ indicates that animals are alive.

Now, the instance statement $S \circ\!\!\to P$ is defined through inheritance as $\{S\} \to P$, while the property statement $S \to \circ\!\!P$ is defined as $S \to [P]$. Since fully reducible to inheritance, instance and property copulas result in no new inference rules.

## 2.5 Compound terms

So far, only atomic terms have been analyzed. However, all aforementioned inference rules operate on *compound* terms as well. A compound term consists of a *connector* and one or more (atomic or compound) terms.

**Definition 2.8.** A compound term has the form of $\{T_1\ con\ T_2\ con \ldots con\ T_n\}$, where *con* is the connector[2], and $T_1 \ldots T_n$ are terms, $n \geq 1$ (Wang, 2006, 2013).

It is often useful to have the means for expressing the compound term's complexity.

**Definition 2.9.** *Syntactic complexity* of a compound term is defined as 1 plus syntactic complexities of its components, where the complexity of an atomic term is 1 (Wang, 2006, 2013).

According to the experience-grounded semantic, the meaning of a term $T$ is defined through its relations with other terms in the system's experience. Now, with the addition of compound terms, this definition is extended. The presence of a compound term $\{T_1\ con \ldots con\ T_n\}$ contributes to the meaning of its components $T_1 \ldots T_n$. Generally speaking, the compound term itself represents a completely new construct, and its meaning cannot be simply reduced to the meanings of individual components.

*NAL-3* recognizes four connector types: *extensional intersection* ($\cap$), *intensional intersection* ($\cup$), *extensional difference* ($-$), and *intensional difference* ($\ominus$). For two terms $T_1$ and $T_2$ the connectors are defined, respectively (Wang, 2006, 2013):

$$(\forall x)((x \to (T_1 \cap T_2)) \Leftrightarrow ((x \to T_1) \wedge (x \to T_2))) \tag{2.26}$$
$$(\forall x)(((T_1 \cup T_2) \to x) \Leftrightarrow ((T_1 \to x) \wedge (T_2 \to x))) \tag{2.27}$$
$$(\forall x)((x \to (T_1 - T_2)) \Leftrightarrow ((x \to T_1) \wedge \neg(x \to T_2))) \tag{2.28}$$
$$(\forall x)(((T_1 \ominus T_2) \to x) \Leftrightarrow ((T_1 \to x) \wedge \neg(T_2 \to x))) \tag{2.29}$$

In addition to inference rules discussed previously, *composition* is introduced to exploit these four connectors. The rule is used to build new compound terms, and, by doing so, to summarize the system's experience. This is one concrete example of how the concept of insufficient knowledge and resources is built into NAL: the logic employs composition in order to reduce the sheer amount of statements in its experience, and enable more efficient reasoning.

[2]Infix notation can be used as well: $\{con\ T_1\ T_2\ \ldots\ T_n\}$

## 2.6 Relational terms

In order to support arbitrary relations between terms, *NAL-4* first introduces additional connector, *product* ($\times$). This connector defines inheritance among individual components of a compound term (Wang, 2006, 2013):

$$((S_1 \times \ldots \times S_n) \rightarrow (P_1 \times \ldots \times P_n)) \Leftrightarrow ((S_1 \rightarrow P_1) \wedge \ldots \wedge (S_n \rightarrow P_n)) \quad (2.30)$$

**Definition 2.10.** A *relational term* $R$ is defined as an atomic term related to a product term by inheritance: $(T_1 \times T_2) \rightarrow R$ or $R \rightarrow (T_1 \times T_2)$ (Wang, 2006, 2013).

For example, the sentence *cat eats bird* can be written as $(cat \times bird) \rightarrow eats$. Here, *eats* is the new relation between *cat* and *bird*.

However, some *structural transformations* of these kinds of sentences are often required. For example, by applying deduction to premises $(cat \times bird) \rightarrow eats$ and $tiger \rightarrow cat$, the system should be able to conclude that $(tiger \times bird) \rightarrow eats$. Currently, this is not feasible since the rule is not able to recognize *cat* as the subject of the first premise. Therefore, it is necessary to define two new constructs, *extensional image connector* ($/$) and *intensional image connector* ($\backslash$), respectively (Wang, 2006, 2013):

$$((T_1 \times T_2) \rightarrow R) \Leftrightarrow (T_1 \rightarrow (/R \diamond T_2)) \Leftrightarrow (T_2 \rightarrow (/R\, T_1 \diamond)) \quad (2.31)$$

$$(R \rightarrow (T_1 \times T_2)) \Leftrightarrow ((\backslash R \diamond T_2) \rightarrow T_1) \Leftrightarrow ((\backslash R\, T_1 \diamond) \rightarrow T_2) \quad (2.32)$$

Here, the $\diamond$ symbol marks the position of either $T_1$ or $T_2$. The first premise can now be rewritten as $cat \rightarrow (/eats \diamond bird)$. This structural transformation enables the system to conclude that $tiger \rightarrow (/eats \diamond bird)$, or, when rewritten, that $(tiger \times bird) \rightarrow eats$.

## 2.7 Handling inconsistencies

Unlike forward inference rules discussed so far, a *local* rule accepts two or more statements of the same form, and produces a conclusion with the same form, but a different truth-value. The main purpose of a local rule is to handle knowledge base inconsistencies (Wang, 1994, 2006, 2013). Two statements are said to be inconsistent if they have the same subject, copula, and predicate, but different truth-values. This situation might occur if, for example, the statements are derived from different evidential bases, or by using different inference steps.

There are two local inference rules that deal with inconsistencies: *revision* and *choice* (Wang, 1994, 2006, 2013). Revision is used when the two statements are derived from disjoint evidential sets, while choice is used when the sets share some elements. Given two inconsistent premises with truth-values $\langle f_1, c_1 \rangle$ and $\langle f_2, c_2 \rangle$, revision employs the following functions to calculate the merged statement's truth-value (Wang, 2006, 2013):

$$f = (f_1 c_1(1 - c_2) + f_2 c_2(1 - c_1))/(c_1(1 - c_2) + c_2(1 - c_1)) \tag{2.33}$$

$$c = (c_1(1 - c_2) + c_2(1 - c_1))/(c_1(1 - c_2) + c_2(1 - c_1) + (1 - c_1)(1 - c_2)) \tag{2.34}$$

In case of choice, the conclusion with higher confidence is selected, because it is supported by more evidence. Keeping track of which evidence contributed to which of the two conclusions would be highly impractical, and would severely limit the system's performance.

## 2.8 Answering questions

Choice is also used to select between competing answers to *questions*.

**Definition 2.11.** NAL questions are statements specified using one of the two forms: "? *copula* $P$" and "$S$ *copula* ?", or "$S$ *copula* $P$" (Wang, 2006, 2013).

For the first form, the system is supposed to return the best possible candidate for "?". If, on the other hand, the question is posed using the second form, the system should return the truth-value of the statement that matches it.

However, the knowledge base might contain multiple answers for the same question. For example, the question ? $\rightarrow$ *cat* might be answered using sentences $tiger \rightarrow cat\langle 1.0, 0.8\rangle$ and $lion \rightarrow cat\langle 0.9, 0.9\rangle$. Here, the choice rule selects the answer with higher *expectation* of frequency.

**Definition 2.12.** The expectation of frequency, $e$, specifies how likely is the value of frequency to be confirmed in the near future. It takes the value in $(0, 1)$ and is calculated as $e = (f - 1/2)c + 1/2$ (Wang, 2006, 2013).

Higher expectation values indicate that the answer is supported by more positive and less negative evidence, while the middle value (i.e. $e = 0.5$) indicates that the system has either equal amount of positive and negative evidence, or knows little about the statement. By relying on the expected frequency value, in the above example the system will select *tiger* as the best answer to the posed question.

When working with compound terms, another factor, *syntactic simplicity* of the compound terms is also taken into account.

**Definition 2.13.** Syntactic simplicity $s$ is calculated as $s = 1/n^r$, where $n$ is syntactic complexity of the term, as defined earlier, and $r > 0$ is a system parameter (Wang, 2006, 2013).

If the two competing answers have the same expectation, the simpler answer is chosen (Rodriguez-Fernandez, 1999; Wang, 2013). If the expectations differ, the answer with higher product of $e$ and $s$ is selected.

As shown, choice is used to find the best possible answer to the posed question. However, the answer might not be directly included in the system's experience. *Backward* inference rules are applied in these situations (Wang, 2006,

2013). A backward inference rule accepts a question $Q$ and a premise $P$. It then constructs another question, $Q'$, such that a forward inference rule on $Q'$ and $P$ produces an answer to the original question $Q$. Since inference rules in NAL are reversible (i.e. a premise and the conclusion can be used to produce the other premise), the derived question $Q'$ is obtained by applying a forward inference rule to the original question $Q$ and the premise $P$.

## 2.9 Summary

Non-Axiomatic Logic (NAL) is a term logic with a well-defined language, the experience-grounded semantics, and a set of inference rules for deriving new knowledge and answering questions. It serves as an underlying formalisms for systems that operate under the notion of insufficient knowledge and resources. For example, in NAL the knowledge base does not have to be consistent at all times, and its revision rule is used to handle inconsistencies. Also, if the amount of statements becomes too large, the composition rule can be used to combine them and enable more efficient reasoning. These properties make NAL a good starting point for realizing concrete, practical reasoning systems.

Tables 2.2 and 2.4 represent the summary of forward and compositional inference rules discussed in this chapter. They can be used as a quick reference point for implementations. As a convenience, truth-value functions for syllogistic rules are shown in Table 2.3. For all compositional rules, confidence of the conclusion is calculated as $c_1c_2$. Frequency of the intersection is expressed as $f_1f_2$, of the union as $1 - ((1 - f_1)(1 - f_2))$, and of the difference as $f_1(1 - f_2)$ (Wang, 2013).

As shown in Chapter 5, the first four NAL layers discussed in this chapter provide sufficient theoretical foundation for developing systems that can solve concrete practical problems.

The remaining 5 layers introduce higher-order statements, variable terms, events, time, and operations, and include (so far) theoretical discussions on self-monitoring and consciousness. These layers, however, are not required for the goal of this thesis and will not be discussed. For more information see e.g. (Wang, 2006, 2013), as well as other work of (primarily) dr Pei Wang[3].

The next part of this thesis includes discussions on and concrete proposals of two systems: a distributed non-axiomatic reasoning system that provides agents with cognitive abilities, and a multiagent middleware suitable for deploying these agents in a wide range of web and enterprise applications.

[3] `http://www.cis.temple.edu/~pwang/`, retrieved on August 12, 2014.

Table 2.2: Summary of syllogistic forward inference rules, which accept two premises and derive a conclusion: $\{P_1\langle f_1, c_1\rangle, P_2\langle f_2, c_2\rangle\} \vdash C\langle f, c\rangle$. The top-most row contains $P_2$ statements, while the left-most column contains $P_1$ statements (Wang, 2006, 2013).
Tabela 2.2: Pregled silogističkih pravila izvođenja unapred, koja prihvataju dve premise i izvode zaključak: $\{P_1\langle f_1, c_1\rangle, P_2\langle f_2, c_2\rangle\} \vdash C\langle f, c\rangle$. Gornji red sadrži $P_2$, dok leva kolona sadrži $P_1$ rečenice (Wang, 2006, 2013).

| | $S \to M\langle f_2, c_2\rangle$ | $S \leftrightarrow M\langle f_2, c_2\rangle$ | $M \to S\langle f_2, c_2\rangle$ |
|---|---|---|---|
| $M \to P\langle f_1, c_1\rangle$ | $S \to P\langle F_{ded}\rangle$ | $S \to P\langle F_{ana}\rangle$ | $S \to P\langle F_{ind}\rangle$<br>$S \leftrightarrow P\langle F_{cmp}\rangle$ |
| $M \leftrightarrow P\langle f_1, c_1\rangle$ | $S \to P\langle F_{ana'}\rangle$ | $S \leftrightarrow P\langle F_{res}\rangle$ | $P \to S\langle F_{ana'}\rangle$ |
| $P \to M\langle f_1, c_1\rangle$ | $S \to P\langle F_{abd}\rangle$<br>$S \leftrightarrow P\langle F_{cmp}\rangle$ | $P \to S\langle F_{ana}\rangle$ | |

Table 2.3: Summary of truth-value functions for syllogistic forward inference rules shown in Table 2.2. Function $\langle F_{ana'}\rangle$ is equal to $\langle F_{ana}\rangle$ with the reversed order of premises (Wang, 2006, 2013).
Tabela 2.3: Pregled funkcija za istinitosne vrednosti silogističkih pravila izvođenja unapred iz Tabele 2.2. Funkcija $\langle F_{ana'}\rangle$ je ekvivalentna funkciji $\langle F_{ana}\rangle$ sa obrnutim redosledom premisa (Wang, 2006, 2013)

| Function | Frequency | Confidence |
|---|---|---|
| $\langle F_{ded}\rangle$ | $f_1 f_2$ | $f_1 f_2 c_1 c_2$ |
| $\langle F_{ind}\rangle$ | $f_1$ | $\frac{f_2 c_1 c_2}{f_2 c_1 c_2 + k}$ |
| $\langle F_{abd}\rangle$ | $f_2$ | $\frac{f_1 c_1 c_2}{f_2 c_1 c_2 + k}$ |
| $\langle F_{cmp}\rangle$ | $\frac{f_1 f_2}{f_1 + f_2 - f_1 f_2}$ | $\frac{(f_1 + f_2 - f_1 f_2) c_1 c_2}{(f_1 + f_2 - f_1 f_2) c_1 c_2 + k}$ |
| $\langle F_{ana}\rangle$ | $f_1 f_2$ | $f_2 c_1 c_2$ |
| $\langle F_{res}\rangle$ | $f_1 f_2$ | $(f_1 + f_2 - f_1 f_2) c_1 c_2$ |

Table 2.4: Summary of compositional forward inference rules. Note that $T_1$ and $T_2$ need to be different, and should not contain each other as components (Wang, 2006, 2013).
Tabela 2.4: Pregled kompozicionih pravila izvođenja unapred. Napomena: termovi $T_1$ i $T_2$ moraju biti različiti, i jedan term ne sme predstavljati komponentu drugog (Wang, 2006, 2013).

| | $M \to T_1\langle f_1, c_1\rangle$ | $T_1 \to M\langle f_1, c_1\rangle$ |
|---|---|---|
| $T_2 \to M\langle f_2, c_2\rangle$ | | $(T_1 \cup T_2) \to M\langle F_{int}\rangle$<br>$(T_1 \cap T_2) \to M\langle F_{uni}\rangle$<br>$(T_1 \ominus T_2) \to M\langle F_{dif}\rangle$<br>$(T_2 \ominus T_1) \to M\langle F_{dif'}\rangle$ |
| $M \to T_2\langle f_2, c_2\rangle$ | $M \to (T_1 \cap T_2)\langle F_{int}\rangle$<br>$M \to (T_1 \cup T_2)\langle F_{uni}\rangle$<br>$M \to (T_1 - T_2)\langle F_{dif}\rangle$<br>$M \to (T_2 - T_1)\langle F_{dif'}\rangle$ | |

# Part II

# DNARS and Siebog

# An overview of Part II

Part II represents the main part of this thesis, and discusses its main contributions. It consists of three chapters.

In Chapter 3, we propose an architecture of a new reasoning system based on the Non-Axiomatic Logic, named *Distributed Non-Axiomatic Reasoning System* (DNARS). The main advantage of DNARS, when compared to all other existing reasoning and cognitive architectures, is that it leverages state-of-the-art techniques for large-scale, distributed data management and processing. This approach allows DNARS to operate on top of very large knowledge bases, while serving large numbers of external clients with real-time responsiveness.

The overall architecture of DNARS, presented in Section 3.2, consists of *inference engines* for answering questions and deriving new knowledge, and a *backend knowledge base* for storing the system's knowledge and experience.

DNARS uses a graph-based representation of knowledge. Therefore, Section 3.3 analyses state-of-the-art approaches for large-scale graph processing, while Section 3.4 discusses how the knowledge base in DNARS is modeled using this graph-based approach. In Sections 3.5 and 3.7 we propose two possible concrete realizations of this model, along with appropriate algorithms for inference processes. In addition, Section 3.6 presents how external clients can be notified of changes in the system's knowledge base in order to act accordingly.

In the next chapter of this part (Chapter 4) we propose a new multiagent middleware named *Siebog*. Siebog is an enterprise-scale multiagent middleware designed to support agents in web environments, but in accordance to the modern standards. It successfully combines features of clustered computing on the server and cross-platform execution on the client side, in order to provide agent load-balancing, fault-tolerance, true platform-independence, heterogeneous mobility, cross-platform messaging etc.

During the development of Siebog, a strong emphasis has been put on standards compliance. All of its internal components, including agents, can easily interact with, or be integrated into existing web and enterprise software systems. For example, Siebog agents can publish their functionalities in form of web services, can easily invoke other enterprise components, or perform object-relational mapping. Therefore, Siebog may help in bridging the gap between the agent technology and industrial, non-agent-based software systems.

The main benefits of Siebog are summarized in Section 4.1. The architec-

ture of Siebog's server-side component, its characteristics and functionalities, is described in Section 4.2. The client-side features of Siebog are described in Section 4.3. Section 4.4 discusses how the integrated, unified Siebog framework can additionally provide heterogeneous mobility, cross-platform messaging, and code sharing. Finally, Section 4.5 describes the process of extending Siebog with the support for DNARS-based intelligent agents.

In the last chapter in this part (Chapter 5) we present several case studies that confirm some important assertions made throughout the thesis. First, Section 5.1 shows how the combination of enterprise and web standards in Siebog can be used to develop an agent-based application that mimics the behavior of modern content-sharing networks. Section 5.3 evaluates the runtime performance of the DNARS' Resolution engine, and demonstrates its efficiency in practice. Finally, Section 5.4 presents how DNARS-based intelligent Siebog agents can be used to solve a particular problem of deriving new structured knowledge.

# Chapter 3

# Distributed Non-Axiomatic Reasoning System

This chapter presents a novel general-purpose reasoning architecture named *Distributed Non-Axiomatic Reasoning System* (DNARS). DNARS is built on top of the NAL formalism (Wang, 2013), and on the general guidelines for non-axiomatic reasoning (Wang, 2006). It incorporates an efficient knowledge management system, and a set of inference engines for answering questions and deriving new knowledge.

## 3.1 Motivation and main features of DNARS

The term *Big Data* is used to describe large quantities of highly diverse information, often collected at high frequencies, which traditional approaches cannot efficiently process and analyze (Baesens, 2014). In recent years, the need for Big Data analytics has found its place in a wide range of industrial applications, and is often seen as a crucial asset in the world that produces an ever increasing amounts of information (Simon, 2012).

The overall need for Big Data analytics has resulted in a number of concrete, practical technologies, including *NoSQL* and *Graph databases* (Robinson et al., 2013; Sadalage and Fowler, 2012; Tiwari, 2011), the *MapReduce* programming model (Dean and Ghemawat, 2008; White, 2012), etc.

For an artificial intelligence system, the ability to efficiently process large amounts of knowledge is one of the key requirements (Hovy et al., 2013). DNARS is designed to exhibit this property, for which it relies on a number of Big Data concepts. For example, DNARS includes a distributed, highly-scalable backend knowledge base, which also features fault-tolerance through data replication.

In order to work with these kinds of knowledge bases, DNARS includes a set of algorithms that realize NAL inference rules in an efficient manner. As a result, DNARS can provide thousands of answers per second from a knowledge base of over 75 million statements (see Section 5.3).

NAL is used as the underlying formalism in DNARS because its "philosophy" fits nicely into the DNARS' overall goals. As discussed previously, the logic is built around the concept of insufficient knowledge and resources. Unlike many other formalisms for intelligent agents (e.g. Guerra-Hernandez et al., 2009; Mora et al., 1999; Singh et al., 1999), NAL can work with inconsistent knowledge bases and use the revision and choice rules to handle these inconsistencies. In line with this feature, the NoSQL database used in DNARS is designed to temporarily sacrifice data consistency in order to achieve high availability (Gilbert and Lynch, 2002; Hewitt, 2010).

The overall functionalities of DNARS' inference engines are directed by NAL inference rules and by discussions in (Wang, 2006, 2013). The novelty in DNARS are the algorithms (Sections 3.5 and 3.7) that enable these engines to operate in highly-scalable environments. Furthermore, the overall architecture of DNARS (Section 3.2) and its organization of the backend knowledge base (Section 3.4, along with Section 3.6), is what distinguishes DNARS from other existing artificial intelligence systems.

## 3.2 General overview of the architecture

General architecture of DNARS is outlined in Fig. 3.1. The main components of the proposed system are:

- *Resolution engine*: answers client's questions.
- *Forward inference engine*: derives new knowledge.
- *Short-term memory*: contains statements relevant to the active processing cycles, and problems that need to be solved.
- *Knowledge domain*: a sub-set of the overall knowledge base, containing mutually dependent or related statements.
- *Backend knowledge base*: the system's overall knowledge base, representing its entire experience.
- *Event manager*: a handler for events generated by changes in (parts of) the knowledge base.

Components are broadly organized into two categories. Resolution and Forward inference engines are referred to as *DNARS Inference engines*, while the remaining components are described as the *Backend knowledge base*. Each external client is associated with one set of inference engines, but there is only a single Backend knowledge base for all of them. However, the knowledge base is designed to be highly scalable, and can be partitioned to support multiple isolated and cooperating clients.

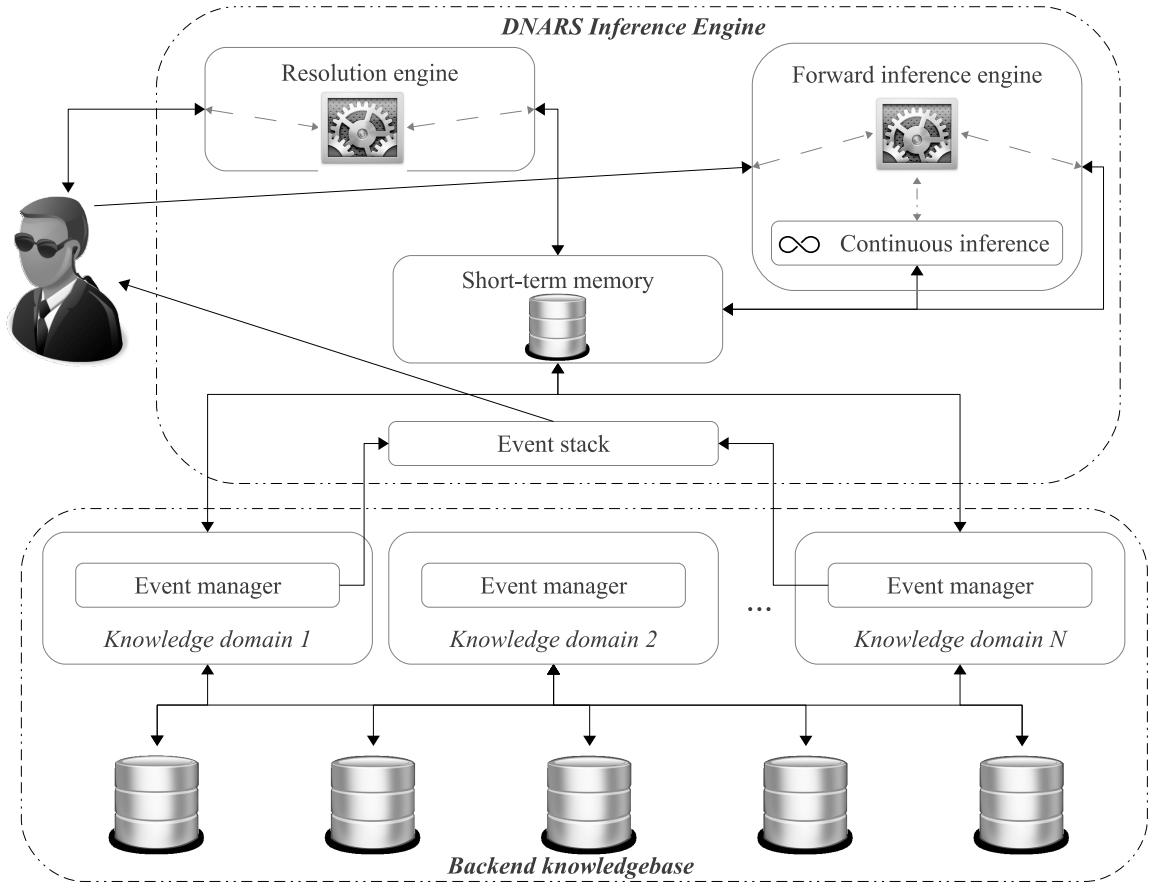



Figure 3.1: General architecture of the proposed Distributed Non-Axiomatic Reasoning System and the organization of knowledge.
Slika 3.1: Opšta arhitektura predloženog Distribuiranog Sistema za Ne-Aksiomatsko Rasuđivanje i organizacija znanja.

### 3.2.1 DNARS inference engines

*Resolution* and *Forward inference engines* are used to, respectively, answer questions and derive new knowledge. Therefore, they represent the core of DNARS' inference capabilities. As discussed in Chapter 2, two types of questions are supported:

- Questions containing "?", i.e. "*S copula* ?" or "? *copula P*". In this case, the Resolution engine inspects the system's knowledge base and finds the best substitute for "?".
- Questions in the form of "*S copula P*". The answer to this question is a statement "*S copula* $P\langle f, c\rangle$". If the answer is not directly available in the system's knowledge base, the engine will try to derive it using backward inference rules.

In general, the Resolution engine should provide answers in real time for the first type of questions. For the second type, the answer is given in real time only if it is directly available in the system's knowledge base. Otherwise, the backward inference process is started in the background and the client is notified of the solution later on.

The Forward inference engine provides direct implementation of forward inference rules defined by NAL. It operates in *inference cycles*, as follows. The execution of each cycle is triggered either by an external client or by an internal process. In case of the former, input statements provided by an external client are used to load all *relevant* statements from the system's knowledge base. Relevant statements for the input *S copula* $P\langle f, c\rangle$ are statements that have $S$ or $P$ as subject or predicate. The unified set of input and relevant statements serves as a starting point for forward inference rules, which may produce another set of conclusions. As the final step of the inference cycle, input, relevant statement and conclusions are merged together, any inconsistencies are resolved, and the final output is stored back in the system's knowledge base.

However, the Forward inference engine can also operate on its own. When idle and not receiving any new input from external clients, the engine will select a (random) statement from the knowledge base and use it as the input. As noted by (Wang, 2013), this continuous internally-triggered inference is one of the main differences between inference engines and advanced knowledge retrieval systems.

### 3.2.2 Backend knowledge base overview

One of the main design goals for DNARS is to develop a system that can efficiently handle large quantities of knowledge[1]. Therefore, the Backend knowledge base of DNARS is designed as a distributed, scalable architecture that consists of three layers. At the bottom-most layer, the entire knowledge base is physically

[1] In today's terms, *large quantities of knowledge* refer to terabytes or petabytes of data.

partitioned and distributed across a number of machines. It uses *horizontal scaling*: as the amount of data increases, the runtime performance is maintained by simply adding more processing nodes to the underlying cluster (Michael et al., 2007). This design approach has two main benefits:

- It enables the backend storage to manage large amounts of data. By introducing proper data distribution rules, faster lookups and retrievals of relevant statements can be achieved.
- It provides fault-tolerant features, as the data is replicated across cluster nodes. There is no single point of failure, and the knowledge base can remain intact in case of hardware and software failures.

On top of this layer, the entire knowledge base is organized into one or more *Knowledge domains*. Domains partition the system's knowledge base into distinct categories. This enables the system to work with and focus on a subset of its knowledge. At runtime, one or more domains can be consulted. This organization also supports the multi-client nature of DNARS. The knowledge belonging to one external client can be stored in a separate domain. However, many clients can also work with same domains (one or more), and, by doing so, exhibit cooperated behavior through knowledge and experience sharing. As shown in Fig. 3.1, the content of one domain can be physically distributed across many machines of the bottom layer, and can be intermingled with the content of other domains. The task of properly storing and retrieving the domain data is delegated to the bottom layer.

Finally, the *Short-term memory* (STM) module is placed at the top layer. This is the knowledge base directly available to DNARS inference engines, and represents the basis for their inference cycles. The main purpose of STM is to serve as the optimization module. Its content should entirely fit in the runtime memory of the host machine, acting as a fast in-memory storage. Once a set of related inference cycles is completed, the STM content is merged back into corresponding (again, one or more) domains.

It is important for clients, especially in cooperative, domain sharing mode, to learn about changes in the knowledge base. For example, the client may wish to perform certain actions in response to newly derived conclusions. To support this feature, DNARS incorporates the *Event manager* module. A change in a knowledge domain will generate one or more *events*, which will be collected by the manager and delivered to clients connected to the domain. This behavior can be seen as a simulation of the blackboard system used in multiagent cooperation, described earlier. Together with continuous processing of the Forward inference engine, it provides the basis for reactive behavior of intelligent agents (Wooldridge and Jennings, 1995).

Knowledge bases based on NAL statements can be represented by *property graphs*. A property graph is a directed, *multi-relational* graph with any number of properties attached to vertices and edges (Robinson et al., 2013; Rodriguez and Shinavier, 2010). That is, it is a graph which can include different types of edges (e.g. for representing inheritance and similarity), and in which each vertex

or an edge can have any number of *key* → *value* pairs attached to it. Fig. 3.2 shows an example of a set of NAL statements and the corresponding property graph.

In relatively recent times, large-scale analysis and processing of (property) graphs has become especially important, due to increasing demands of modern web applications, such as social networks. Therefore, there exists a range of algorithms and technologies for efficient graph analytics. These solutions provide a strong basis for an efficient realization of the proposed DNARS architecture.

## 3.3 Large-scale graph processing

Large-scale graph analysis and manipulation is a thriving area, with a number of frameworks utilized by both the scientific community and the industry. This section briefly analyses the most popular approaches, and then provides a discussion on their usage in implementing DNARS architecture.

### 3.3.1 NoSQL databases

Transactions in relational database systems have been designed around the concepts of *atomicity*, *consistency*, *isolation*, and *durability* (ACID) (Haerder and Reuter, 1983). Atomicity requires all parts of the transaction to succeed, or the entire transaction fails. Consistency ensures that the transaction leaves the data in a consistent state, according to predefined constraints, relations, etc. Isolation ensures that transactions don't depend on each other, and that a set of transactions can be applied sequentially or in parallel. Finally, durability indicates that the results of a transaction need to remain permanent.

Over the years, relational databases have become the most widely used storage systems, to the great part due to these characteristics. However, in recent years, they have proven to be inadequate for handling large quantities of unstructured and interconnected information, such as those generated by modern web applications (Robinson et al., 2013; Sadalage and Fowler, 2012; Tiwari, 2011). For example, relational databases require a well-defined model of the data, which these applications cannot easily define. In addition, they do not scale well enough to handle large volumes of data or numbers of concurrent users.

To alleviate these and other issues, a new model of a database, named NoSQL[2] has been proposed. NoSQL databases generally operate on top of computer clusters and employ horizontal scaling to meet increasing demands. This is not their defining characteristic, as some NoSQL databases operate on a single machine (for example, *Oracle Berkeley DB*[3]). However, those that do operate in clustered environments can easily provide advanced features, such as fault-tolerance techniques through data replication.

[2]The term *NoSQL* is somewhat inadequate and misleading, since many of these database do employ SQL-like languages. NoSQL is therefore often defined to stand for *Not Only SQL*.

[3]`http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`, retrieved on August 12, 2014.

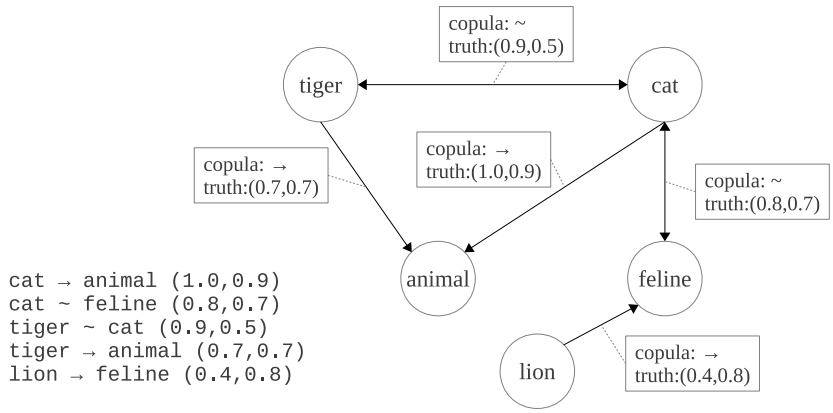



Figure 3.2: A set of arbitrary NAL statements and the corresponding property graph. Note that edges representing similarities are bidirectional, expressing the symmetric nature of the copula.
Slika 3.2: Skup proizvoljnih NAL rečenica i odgovarajući graf osobina. Dvosmerne grane prestavljaju sličnosti, čime se izražava simetričnost ove relacije.

Clustered NoSQL databases are much more complex to manage and, in general, do not exhibit all ACID properties. More concretely, the so-called *CAP theorem* (or, *Brewer's Conjecture*) suggests that distributed systems, including NoSQL databases, cannot fully achieve all three of the following properties (Gilbert and Lynch, 2002):

- *Consistency*: All nodes have the same view of data;
- *Availability*: Every operating cluster node will (eventually) produce the requested response;
- *Partition tolerance*: The system continues to operate even if some cluster nodes fail.

Here, it is important to note that distributed systems do not have to choose exactly two out of the three properties, but can instead achieve one to a lesser and other to a higher degree (Sadalage and Fowler, 2012). Nonetheless, this situation fits nicely into the NAL's assumption of insufficient knowledge and resources, as discussed earlier.

Based on the underlying data model, many categories of NoSQL databases exist, including key-values, column-oriented, document, and graph databases (Tiwari, 2011). Obviously, for DNARS, graph databases are of the special importance.

A graph database includes a concrete storage system, and a processing engine for graph traversals and manipulations. In addition, the so-called *native* graph databases provide *index-free adjacency*, which means that they store adjacency lists directly, instead of relying on index-based lookups (Robinson et al., 2013).

Adjacency lists can be stored in a specifically-built architecture or in an existing database. The most notable implementation of the first approach is *Neo4j*[4], whereas *Aurelius Titan*[5] can store graphs in a number of NoSQL databases.

The graph stored in a NoSQL database can be accessed and processed in several ways, with the three most widely-used approaches described in the remainder of this section.

### 3.3.2 The MapReduce programming model

*MapReduce* is not a graph processing framework *per se*, but a general-purpose programming model for large-scale data analysis and manipulation (Dean and Ghemawat, 2008; White, 2012). However, it can be used as the basis for higher-level graph-oriented algorithms.

The MapReduce model incorporates two sets of functions: *mappers* and *reducers*. A mapper receives a set of key-value pairs as the input, and produces another set of (intermediate) key-value pairs as the output. There can be any number of mappers operating in parallel, on different machines, and processing distinct parts of the initial dataset. Mapper results are collected in runtime memory, periodically written to disk, and distributed among reducers.

A reducer is a functions that receives a key and a set of values as the input, and produces a reduced set of values (e.g. a single value) as the output. Before the reducer can start processing the data, the *sort and shuffle* phase is executed (Dean and Ghemawat, 2008; White, 2012). During this phase, the output of mappers is grouped by keys, with the goal of sending one group to one reducer. Of course, a reducer can process many groups. Since, like mappers, there can be many reducers operating in parallel, on different machines, there can be many end results. These end results can either be merged together to produce the final output, or serve as an input for the next set of mappers and/or reducers.

The entire process is controlled by the *master* node (with mappers and reducers being referred to as *workers*). It keeps track of the dataset partitions, intermediate results, the state of each worker, etc. Since it may represent a single point of failure, the state of the master is periodically stored on a separate machine. However, the general approach is that the master should not fail, and if it does, the entire execution fails (Dean and Ghemawat, 2008).

MapReduce is designed to operate in clusters of commodity hardware: regular, cheap personal computers. Since the cluster can include thousands of these machines, hardware (and software) failures are to be expected. Therefore, fault-tolerance techniques are built into the model. For example, the master node will *ping* its workers at regular time intervals. If no response is received, the worker is assumed to be unavailable, and its jobs are re-assigned.

The most widely-used open-source implementation of MapReduce is *Apache Hadoop*[6] (White, 2012). Hadoop represents an extensive "eco-system" of libraries, tools, and even programming languages that simplify the development

[4] `http://neo4j.org/`, retrieved on August 12, 2014.
[5] `http://thinkaurelius.github.io/titan/`, retrieved on August 12, 2014.
[6] `http://hadoop.apache.org/`, retrieved on August 12, 2014.

of applications based on MapReduce. It is supported by a large base of contributors and end-users.

### 3.3.3 Bulk Synchronous Parallel

*Bulk Synchronous Parallel* (BSP) is a model for parallel computations (Valiant, 1990). It incorporates a number of *components*, each with its own memory and processing capabilities, and capable of communicating with other components. The overall computation is performed in so-called *supersteps*. Within a superstep, a component can receive messages sent to it during the previous superstep, perform a number of local calculations, and send messages to other components. A superstep ends once all components reach a certain *barrier*.

The use of BSP for graph processing has been popularized by by the *Pregel* system (Malewicz et al., 2010). Pregel is a vertex-centric system, which means that graph vertices play the role of BSP components. A user-defined function is attached to vertices and executed in each superstep. Besides sending and receiving messages, the function can modify state of the vertex, its outgoing edge, or even make global changes to the graph.

In Pregel, a vertex is either in *active* or self-imposed *inactive* state. It will remain inactive unless it receives a message, in which case it has to explicitly set its state to inactive again. The overall computation is considered to be finished once all vertices are inactive, and there are no pending messages.

An open-source, Hadoop-based implementation of Pregel is available to the wide community of users in form of the *Apache Giraph* project[7].

### 3.3.4 The TinkerPop stack

The *TinkerPop stack* is a set of standards dedicated to large-scale processing of property graphs, either on a single machine or in a computer cluster[8]. It incorporates the following set of technologies:

- *Blueprints API*: A standard interface for (property) graph databases[9].
- *Pipes*: A data-flow framework[10] based on the *Kahn process network* (Kahn, 1974).
- *Gremlin*: A graph traversal language[11].
- *Frames*: An object-to-graph mapper[12].
- *Furnace*: An extensive higher-level library of graph algorithms[13].

[7] `http://giraph.apache.org/`, retrieved on August 12, 2014.
[8] `http://www.tinkerpop.com/`, retrieved on August 12, 2014.
[9] `https://github.com/tinkerpop/blueprints`, retrieved on August 12, 2014.
[10] `https://github.com/tinkerpop/pipes`, retrieved on August 12, 2014.
[11] `https://github.com/tinkerpop/gremlin`, retrieved on August 12, 2014.
[12] `https://github.com/tinkerpop/frames`, retrieved on August 12, 2014.
[13] `https://github.com/tinkerpop/furnace`, retrieved on August 12, 2014.

- *Rexster*: A server that exposes *Blueprints*-enabled database through *REST* and binary protocols[14].

There are a number of open-source and commercial, full or partial implementations of the TinkerPop stack. One notable open-source implementation is *Aurelius Titan*[15]. As noted earlier, Titan acts as a layer on top of "regular," non-graph-oriented databases, both relational and NoSQL.

MapReduce and its extension, Pregel, represent excellent frameworks for large-scale graphs. However, they cannot easily satisfy all functional requirements of the proposed DNARS architecture. For example, MapReduce is a batch processing framework and, in practice, some time is usually spent on job preparations. Therefore, it might be unsuitable for real-time responsiveness of the Resolution engine. Similarly, these frameworks are best-suited for global, whereas DNARS requires local graph analysis: its inference engines usually start with a single vertex, and then explore its immediate neighbors.

Titan is optimized for local, vertex-centric graph analysis and manipulation. Its additional characteristics include the following:

- Titan is a clustered graph database, which means that it can support all functional requirements of the DNARS Backend knowledge base.
- As noted, Titan can use a number of databases to actually store the graph. Therefore, different backend solution can be tested without affecting the remaining parts of DNARS.
- Besides providing graph storage, Titan implements the entire TinkerPop stack, encompassing a set of standards and algorithms for distributed, large-scale graph processing, and covering many aspects of graph analysis and manipulation.

Given these properties, Titan has eventually been selected as the framework for realizing DNARS. Nonetheless, Section 3.7 discusses the approach of implementing DNARS in pure MapReduce. The purpose of this approach is mainly to serve as a case-study.

## 3.4 Backend knowledge base as a graph database

Titan currently operates on top of several databases, including *Apache HBase*[16] and *Apache Cassandra*[17]. Both can be executed on top of a computer cluster, and can easily be integrated with Hadoop (although HBase currently provides better integration). HBase is based on the *BigTable* model (Chang et al., 2008), while Cassandra stores the data as key-values organized in column families (similar to tables in relational databases).

[14] `https://github.com/tinkerpop/rexster`, retrieved on August 12, 2014.
[15] `http://thinkaurelius.github.io/titan/`, retrieved on August 12, 2014.
[16] `http://hbase.apache.org/`, retrieved on August 12, 2014.
[17] `http://cassandra.apache.org/`, retrieved on August 12, 2014.

Regarding the CAP theorem, HBase sacrifices availability in case of high loads, while Cassandra sacrifices consistency (Hewitt, 2010). Possible lack of consistency in Cassandra is more in line with the NAL's assumption of insufficient knowledge and resources. In addition, Cassandra provides lower latency in random read operations (Rabl et al., 2012), which is essential for real-time responsiveness of the Resolution engine. Due to these reasons, DNARS Backend knowledge base is based on Titan over Cassandra.

Titan stores each vertex of a graph as a separate row in the database. Vertex identifier (i.e. a hash) represents the row-key, while individual columns hold vertex properties and edges (along with their properties). This means that the edge is stored twice in the database – once for each vertex. However, this approach increases the system's runtime performance. During the process of loading relevant statements into the Short-Term Memory (STM), DNARS selects vertices representing subjects and/or predicates of input statements and loads them along with their corresponding adjacency lists. Therefore, the actual STM content is graph vertices and their adjacency lists.

Knowledge domains of the DNARS architecture are mapped to Cassandra *keyspaces* (Hewitt, 2010). A keyspace roughly corresponds to a database in the relational model and can have its own set of settings, such as the replication factor related to fault-tolerance.

Once the correct backend system has been chosen and configured, very little needs to be done in order to satisfy functional requirements of the Backend knowledge base. All functionalities and possible issues are efficiently handled by the Titan-Cassandra combination, allowing the development focus to be placed on realizing the remaining parts of DNARS.

## 3.5 Inference based on the TinkerPop stack

The inference process in DNARS is realized as a set of graph traversal and manipulation algorithms. Titan includes an expressive domain-specific language for this purpose, named *Gremlin*. The actual implementation in DNARS uses a third-party Scala-based realization of the language[18].

### 3.5.1 Forward inference engine

Earlier, Table 2.2 summarized NAL's syllogistic forward inference rules. To recall, syllogistic rules accept two premises with a shared term and derive a conclusion using the remaining terms. The concrete rule to be applied is determined by the position of the shared term and the actual copulas in the premises.

For the purpose of realizing the Forward inference engine, all syllogistic rules are organized into groups and each group is implemented as a separate function. For example, in one group, the first premise is $Prem_1 : P \rightarrow M\langle f_1, c_1\rangle$ while the second premise is either $Prem_2 : S \rightarrow M\langle f_2, c_2\rangle$ or $Prem_3 : S \leftrightarrow M\langle f_2, c_2\rangle$. In this case, abduction, comparison and analogy can be applied, as follows:

[18] `https://github.com/mpollmeier/gremlin-scala`, retrieved on August 12, 2014.

$$\{Prem_1, Prem_2\} \vdash \{S \rightarrow P\langle F_{abd}\rangle, S \leftrightarrow P\langle F_{cmp}\rangle\} \tag{3.1}$$

$$\{Prem_1, Prem_3\} \vdash P \rightarrow S\langle F_{ana}\rangle \tag{3.2}$$

Listing 3.1 shows how these three forward inference rules have been realized in the graph-based implementation of DNARS.

Listing 3.1: A function that accepts either $S \rightarrow M\langle f_2, c_2\rangle$ or $S \leftrightarrow M\langle f_2, c_2\rangle$ as the second premise (judgment), uses an existing statement $P \rightarrow M\langle f_1, c_1\rangle$ as the first premise, and produces conclusions shown in Eq. 3.1 and Eq 3.2.

```
def abductionComparisonAnalogy(judgment: Statement): List[Statement] = {
  graph.getV(judgment.pred) match {
    case Some(m) => // m is the shared term
      val incomingEdges = m.inE(Inherit).toList
      incomingEdges.flatMap { e: Edge => inferForEdge(judgment, e) }
    case None =>
      List()
  }
}

def inferForEdge(judgment: Statement, e: Edge): List[Statement] = {
  val p = e.getVertex(Direction.OUT).term
  if (judgment.subj == p) {
    List() // avoid tautologies
  } else if (judgment.copula == Inherit) {
    abduction(p, judgment, e) ::: comparison(p, judgment, e)
  } else {
    analogy(p, judgment, e)
  }
}

def abduction(p: Term, judg: Statement, e: Edge): List[Statement] = {
  val truth = e.truth.abduction(judg.truth)
  val derived = Statement(judg.subj, Inherit, p, truth)
  keepIfValid(derived)
}

def comparison(p: Term, judg: Statement, e: Edge): List[Statement] = {
  val truth = e.truth.comparison(judg.truth)
  val derived = Statement(judg.subj, Similar, p, truth)
  keepIfValid(derived)
}

def analogy(p: Term, judg: Statement, e: Edge): List[Statement] = {
  val truth = e.truth.analogy(judg.truth, false)
  val derived = Statement(p, Inherit, judg.subj, truth)
  keepIfValid(derived)
}
```

The execution sequence of this function can be summarized as follows:

- The expression `graph.getV(judgment.pred)` selects the vertex that corresponds to the input judgment's predicate (denoted here as `m`), since this is the shared term in the given three syllogistic rules.
- The expression `m.inE(Inherit)` loads all incoming edges for the shared term. Each edge, along with its source and target vertices, represents the existing statement to be used as the first premise. Therefore, a helper function `inferForEdge` is called for each edge.
- The helper function first retrieves the source vertex for the given edge (denoted here as `p`). That is, in the function `inferForEdge` the entire first premise $P \rightarrow M\langle f_1, c_1\rangle$ is retrieved.
- Finally, based on the input judgment's copula, the new conclusions are derived through abduction and comparison, or through analogy. To avoid generating grammatically incorrect statements, the validity of each conclusion is also checked.

All remaining forward inference rules are realized in a similar pattern. As a concrete example of the forward inference process, Listing 3.2 shows a starting knowledge base comprised of three inheritance and two similarity statements. The graph representation of this knowledge base is shown in Fig. 3.3(a).

Listing 3.2: Starting knowledge base of the forward inference example.

```
tiger ↔ cat ⟨0.9, 0.5⟩
tiger → animal ⟨0.7, 0.7⟩
cat → animal ⟨1.0, 0.9⟩
cat ↔ feline ⟨0.8, 0.7⟩
lion → feline ⟨0.4, 0.8⟩
```

The forward inference process starts as the system receives a new input judgment, $cat \rightarrow mammal\langle 1.0, 0.9\rangle$ and adds it to the knowledge base (Fig. 3.3(b)). In this scenario, three forward inference rules are applicable: induction, extensional comparison, and analogy. They are discussed in Chapter 2 and summarized here, respectively:

$$\{M \rightarrow P\langle f_1, c_1\rangle, M \rightarrow S\langle f_2, c_2\rangle\} \vdash S \rightarrow P\langle f, c\rangle \tag{3.3}$$

$$\{M \rightarrow P\langle f_1, c_1\rangle, M \rightarrow S\langle f_2, c_2\rangle\} \vdash S \leftrightarrow P\langle f, c\rangle \tag{3.4}$$

$$\{M \leftrightarrow P\langle f_1, c_1\rangle, M \rightarrow S\langle f_2, c_2\rangle\} \vdash P \rightarrow S\langle f', c'\rangle \tag{3.5}$$

Since induction and extensional comparison take very similar premises and produce similar conclusions, they belong the the same group and are applied in parallel. When applying forward inference rules, DNARS always takes the first premise from the knowledge base, while the new input represents the second

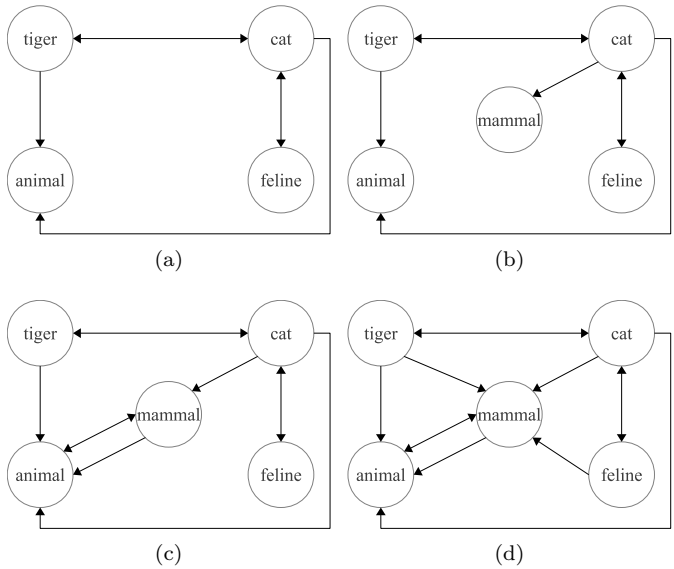



Figure 3.3: An example of a forward inference process in DNARS, from the initial knowledge base (a), after the addition of a new judgment $cat \rightarrow mammal\langle 1.0, 0.9\rangle$ (b), after applying induction and extensional comparison (c), and finally, after applying analogy (d). Unidirectional arrows represent inheritance, while bidirectional arrows represent similarity.
Slika 3.3: Primer procesa izvođenja unapred u DNARS-u, od početne baze znanja (a), nakon dodavanja nove činjenice $cat \rightarrow mammal\langle 1.0, 0.9\rangle$ (b), nakon primene indukcije i proširujućeg poređenja (c), i, konačno, nakon primene analogije (d). Jednosmerne strelice predstavljaju nasleđivanje, dok dvosmerne predstavljaju sličnosti.

premise. In case of induction and extensional comparison, the first execution step determines that the shared term `m` is *cat*. In the second step, the system selects *cat*'s outgoing edges, along with their respective incoming vertices; more concretely, it determines that statements 3 and 4 in Listing 3.2 should be used in place of the first premise. The two inference rules can now be applied, deriving that *mammal is a type of animal* ($mammal \rightarrow animal\langle 1.00, 0.45\rangle$) and that *mammal is animal* ($mammal \leftrightarrow animal\langle 1.00, 0.45\rangle$). The new knowledge base, i.e. with two new conclusions, is shown in Fig. 3.3(c).

In case of analogy, the shared term `m` is also *cat*. The two existing similarity statements that include this term are $tiger \leftrightarrow cat\langle 0.9, 0.5\rangle$ and $cat \leftrightarrow feline\langle 0.8, 0.7\rangle$. Although the rule 3.5 is directly applicable only to the second premise, DNARS uses the fact that similarity is symmetric by Def. 2.5. Therefore, it automatically transforms the first statement into $cat \leftrightarrow tiger\langle 0.9, 0.5\rangle$ and derives two new conclusions: $tiger \rightarrow mammal\langle 0.90, 0.41\rangle$, and $feline \rightarrow mammal\langle 0.80, 0.50\rangle$. The final knowledge base is shown in Fig. 3.3(d).

### 3.5.2 Resolution engine

The Resolution engine is in charge of answering questions in form of $S \rightarrow ?$ and $? \rightarrow P$. It also needs to perform this task as fast as possible. To accommodate this requirement the knowledge base in DNARS includes edge indexes.

As discussed in Section 2.8, if there are multiple answers to a question, the choice rule is used to select the answer with the higher expectation of frequency, $e = (f - 1/2)c + 1/2$. If two terms have the same expectation, the rule considers syntactic simplicity of terms, $s = 1/n^r$, where $n$ is syntactic complexity of the term, and $r > 0$ is a system parameter. The syntactic complexity is further defined to be 1 for atomic terms or 1 plus complexities of compound term's components. If two answers have the same expectation, the simpler one is chosen (Wang, 2006, 2013).

DNARS encodes these expressions into a numeric value that represents the edge index. More concretely, an edge between vertices $S$ and $P$ has two indexes: one including the expectation and the simplicity of $S$, and one including the expectation and the simplicity of $P$. When posed a question, for example $S \rightarrow ?$, the Resolution engine sorts all candidate answers $C$ by the indexes of edges that come out of $S$ and into $C$, and then returns the best one (or $n$ best ones).

The use of indexes speeds up the Resolution engine's execution significantly. It, however, may slow down the forward inference, the indexes need to be updated as edges are added or as the truth-value of an existing edge is changed. However, there are no strict time constraints for the forward inference (Wang, 2006) and it is usually executed in the background, so this is a suitable trade-off.

The question answering process of the DNARS' Resolution engine is shown in Listing 3.3. The main function (`answer`) accepts the question and the desired number of answers and returns a list of terms that fit the missing element. It relies on two helper functions, `getBestSubjects` for $? \rightarrow P$ and `getBestPredicates` for $S \rightarrow ?$.

The execution sequence of the helper function `getBestPredicates` can be summarized as follows. First, the expression `getV(subj)` returns the vertex corresponding to the question's known term $S$. If this vertex does not exist in the knowledge base, there are no possible answers. Otherwise:

- Keep only the edges that match the question's copula;
- Out of those, keep only the edges that come out of the known term;
- Sort them in a descending order by the value that encodes the expectation of frequency and the syntactic simplicity of the missing predicate term;
- Keep only the first *limit* edges and get their target vertices.

Listing 3.3: The question answering process performed by the DNARS' Resolution engine.

```
def answer(question: Statement, limit: Int = 1): List[Term] = {
  if (question.subj == Question) {
    getBestSubjects(question.pred, question.copula, limit)
  } else if (question.pred == Question) {
    getBestPredicates(question.subj, question.copula, limit)
  } else {
    throw new IllegalArgumentException("Invalid question format.")
  }
}

def getBestPredicates(subj: Term, copula: String, limit: Int):
    List[Term] = {
  getV(subj) match {
    case Some(v) =>
      val vertices = v.asInstanceOf[TitanVertex].query()
        .labels(copula)
        .direction(Direction.OUT)
        .orderBy("predExp", Order.DESC)
        .limit(limit)
        .vertices()
      iterableToList(vertices)
    case None =>
      List()
  }
}
```

As a concrete example, Listing 3.4 shows a knowledge base of five statements describing a cat. Fig. 3.4(a) shows how these statements are stored in the graph. When the system receives the question *cat* $\rightarrow$ ?, the Resolution engine will:

- Exclude *tiger*, as it is related to *cat* through similarity;
- Exclude *fluffy*, as in this relation *cat* represents the target vertex;
- Sort the remaining edges as described previously. The resulting graph is shown in Fig. 3.4(b); and
- Keep only the first edge, and return *feline* as the best possible answer.

Listing 3.4: Initial knowledge base of five statements describing a cat.

```
fluffy → cat ⟨1.0, 0.9⟩
cat ↔ tiger ⟨1.0, 0.9⟩
cat → mammal ⟨0.6, 0.3⟩
cat → feline ⟨1.0, 0.9⟩
cat → animal ⟨0.6, 0.4⟩
```

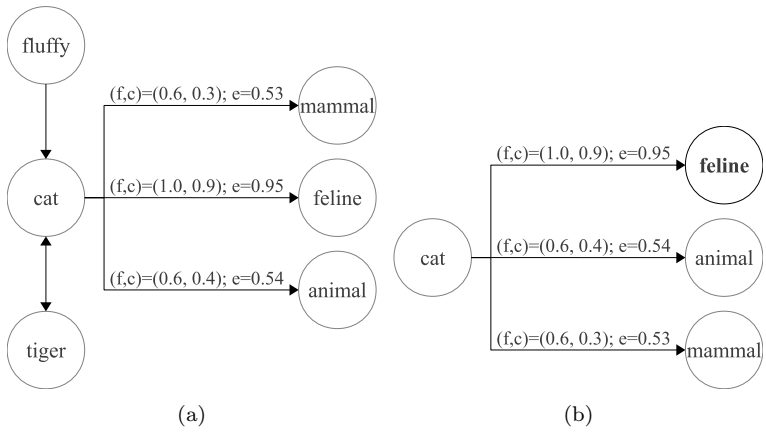



Figure 3.4: For the question $cat \rightarrow ?$, the best answer is $feline$. Knowledge base initially includes three *cat*-related statements (a), which are sorted during the question answering process, and according the coresponding edges' indexes (b).
Slika 3.4: Najbolji odgovor na pitanje je $cat \rightarrow ?$ je $feline$. Baza znanja na početku sadrži tri rečenice koje opisucu mačku (*cat*) (a), koje se sortiraju prilikom traženja odgovora, a na osnovu indeksa odgovarajućih grana u grafu (b).

The Resolution engine is also in charge of performing backward inference. In this case, it accepts a question in form of $S \rightarrow P$. If the answer is not directly available in the system's knowledge base, it will try to derive it using the backward inference process described in Section 2.8. Since this process can take longer time to complete, it will be performed asynchronously, and the client will be notified of the result through the Event stack, described next.

## 3.6 Event manager

Event manager in DNARS is designed using the well-known *Observer* design pattern (Purdy and Richter). In this pattern, the *subject* maintains a list of *observers*, and notifies them of state changes. The Observer pattern is most commonly used in event notifications; for example, in the Java *Swing* GUI library, observers are built by implementing corresponding listener interfaces.

The internal functioning of the Event manager is shown in Fig. 3.5. Each Knowledge domain has a single Event manager associated with it, and the domain publishes descriptions of changes to the manager. At the same time, interested clients are registered to receive notifications from the manager.

The Event manager holds two lists: the list of pending events, and the list of observers. Both lists are directly controlled by internal *Event Dispatch Threads* (EDTs). An EDT polls pending events and notifies all registered observers. It

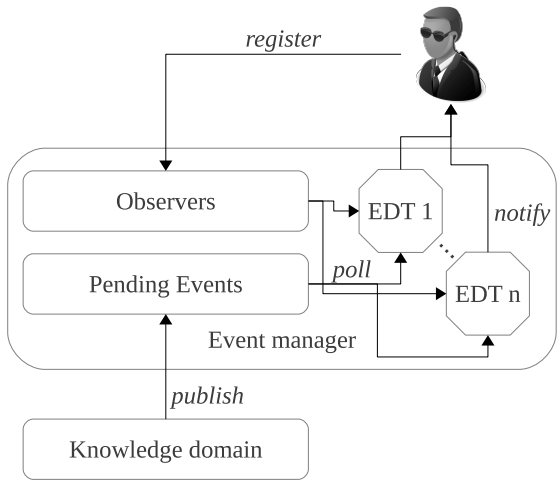



Figure 3.5: Internal organization of the Event manager.
Slika 3.5: Interna organizacija Rukovodioca događajima.

uses simple synchronization primitives in order to prevent data corruption (e.g. missed events), as shown in Listing 3.5.

Listing 3.5: The core functionality of the Event manager and its Event Dispatch Thread.

```
override def run(): Unit = {
  while (!Thread.interrupted) {
    processEvents()
  }
}

def processEvents(): Unit = {
  try {
    waitAndDispatch()
  } catch {
    case _: InterruptedException => Thread.currentThread.interrupt()
    case ex: Exception => LOG.warn("Exception in EDT.", ex)
  }
}

def waitAndDispatch(): Unit = {
  var eventsToDispatch: List[EventPayload] = null
  pendingEvents synchronized {
    waitForEvents()
    eventsToDispatch = cloneOfPendingEvents
  }
  dispatch(eventsToDispatch)
}
```

```
def waitForEvents(): Unit = {
  while (pendingEvents.length == 0) {
    pendingEvents.wait()
  }
}

def cloneOfPendingEvents(): List[EventPayload] = {
  val copy = pendingEvents.toBuffer
  pendingEvents.clear()
  copy.toList
}
```

The actual event dispatching (the `dispatch` function) is designed to be extensible. In order to support intelligent agents in the *Siebog* multiagent middleware, the function actually sends FIPA ACL messages to registered observer agents. These messages are then intercepted by the middleware and delivered to appropriately annotated methods of the agent class, as discussed more details in Section 4.5.

## 3.7 Inference based on MapReduce

Apache Hadoop (an open-source implementation of MapReduce) is currently one of the most widely-used frameworks for large-scale distributed data processing. Here, we propose an alternative set of algorithms for DNARS inference engines that utilize the MapReduce programming model.

Although Hadoop is Java-based, writing *mappers* and *reducers* in Java can be a difficult task. It requires a lot of *boilerplate* code, the use of custom data types, etc. This is why Hadoop supports many different languages and approaches for writing worker functions (White, 2012). Functional programming languages represent a "natural fit" for data processing algorithms. One such language targeting Hadoop is *Scalding*, a domain-specific language based on Scala[19]. Scalding itself further relies on *Cascading*[20], a data-flow framework that acts as an abstraction layer on top of Hadoop.

With Cascading, NAL statements are taken from a *source* (e.g. the knowledge base), flow through one or more *pipes* and are finally written to a *sink* (e.g. back to the knowledge base) (Nathan, 2013). NAL inference rules are expressed as operations on pipes. With Scalding, this approach is abstracted even more, and allows for inference rules to be expressed as functions on regular Scala collections.

Therefore, in the Hadoop-based realization of inference engines, the system's knowledge and experience is handled in form of pipes or collections, while the engines themselves are defined as functions on these pipes. The entire inference process is organized into a number of Scalding *jobs*, which are automatically transformed into Hadoop mappers and reducers.

[19] `https://github.com/twitter/scalding/`, retrieved on August 12, 2014.
[20] `http://www.cascading.org/`, retrieved on August 12, 2014.

### 3.7.1 Forward inference engine

When working with Scalding over Hadoop, it is useful to abandon the graph-based nature of NAL statements, and treat them as independent *tuples* flowing through pipes. A tuple is described using the following set of fields, that respectively correspond to subject, copula, predicate, and frequency-confidence elements of a NAL statement:

Listing 3.6: Description of NAL statements that flow through Cascading pipes.

```
object StatFields extends Enumeration { val s, c, p, fc = Value }
```

Similarly as with the Titan implementation, NAL rules are first organized into logical groups, and then each group is implemented as a separate function. Listing 3.7 shows a function that derives the three conclusions in equations 3.1 and 3.2. The implementation corresponds to previous Titan-based approach shown in Listing 3.1.

Listing 3.7: A function that accepts $P \rightarrow M\langle f_1, c_1\rangle$ as the first, and either $S \rightarrow M\langle f_2, c_2\rangle$ or $S \leftrightarrow M\langle f_2, c_2\rangle$ as the second premise, and produces conclusions shown in Eq. 3.1 and Eq 3.2.

```
def abd_cmp_ana(prem1: Pipe, prem2: Pipe): Pipe = {
  join(p1 -> p2, prem1, prem2)
    .filter(s1, s2, c1) { (s1: Term, s2: Term, c1: Copula) =>
      s1 != s2 && c1 == Inheritance
    }
    .flatMapTo((s1, p1, s2, c2, fc1, fc2) -> StatFields) {
      (s1:Term, p1:Term, s2:Term, c2:Copula, fc1:Truth, fc2:Truth) =>
        c2 match {
          case Inheritance =>
            val con1 = (s2, Inheritance, s1, fc1.abduction(fc2))
            val con2 = (s2, Similarity, s1, fc1.comparison(fc2))
            List(con1, con2)
          case Similarity =>
            List((s1, Inheritance, s2, fc1.analogy(fc2, false)))
        }
    }
}
```

The function accepts two pipes of premises: `prem1` includes existing statements taken from the system's knowledge base, while `prem2` includes new statements. As noted earlier, these new statements can be specified by external clients or chosen randomly during the continuous inference. The two input pipes are first joined on common fields, to produce a temporary third pipe, which is then filtered and transformed according to the inference rules.

In the given listing, fields `s1`, `c1`, `p1`, and `fc1` are used to represent, respectively, subjects, copulas, predicates, and truth-values of statements in the first pipe. In the first execution step, the two input pipes are joined so that state-

ments with a shared term are grouped together. As an example, the next set of expressions represents contents of input pipes and the intermediary joined pipe:

$$
\begin{aligned}
prem1 : & \; cat, \rightarrow, animal, \langle 1.0, 0.9 \rangle \\
& \; water, \rightarrow, liquid, \langle 1.0, 0.9 \rangle \\
& \; animal, \leftrightarrow, mammal, \langle 1.0, 0.9 \rangle \\
prem2 : & \; tiger \rightarrow animal \langle 0.6, 0.5 \rangle \\
joined : & \; cat, \rightarrow, animal, \langle 1.0, 0.9 \rangle, tiger, \rightarrow, \langle 0.6, 0.5 \rangle \\
& \; animal, \leftrightarrow, mammal, \langle 1.0, 0.9 \rangle, tiger, \rightarrow, \langle 0.6, 0.5 \rangle
\end{aligned}
$$

In the second step, the joined pipe is filtered to exclude tautologies and to keep only the statements in which the first premise is inheritance. In the final step, the function `flatMapTo` is used to transform each remaining statements using the three inference rules – abduction, comparison, and analogy.

Possible execution of this Scalding job is shown in Fig. 3.6. The system knowledge base is split among a number of `joinWithTiny` mappers, used by the helper `join` function in the above listing. Mapper `joinWithTiny` is efficient when the second pipe (i.e. the one containing input statements) is *tiny*[21], and can be directly copied to each mapper. Two `joinWithTiny` mappers produce results, sending them to corresponding `filter` mappers. The second filter will discard its pipe, since `c1` is a similarity copula. The output of the first filter is forwarded to the `flatMapTo` mapper, which, finally, produces the two conclusions.

### 3.7.2 Resolution engine

The process of answering question in form of *S copula* ? and ? *copula P* consists of two steps. First, the Resolution engine needs to filter the knowledge base, and to keep only the relevant statements – those that have the known term as subject or predicate. Relevant statements are then transformed into tuples in the form of `(answer, expectation, simplicity)`, where `answer` represents the candidate answer term, while the expectation and simplicity were described in Section 2.8. The source code for this step is shown in Listing 3.8.

Afterwards, the function needs to choose the best possible answer. Here, the `groupAll` function is used, which sends all candidate answers to a single reducer. Although this can be very inefficient if the pipe is large, it is necessary in order to appropriately sort the candidates. In any case, it is not expected that a question will have a large number of possible answers. The answer selection process is shown in Listing 3.9.

[21] According to the Scalding documentation, the adjective *tiny* describes a pipe with up to a couple of thousands of elements.

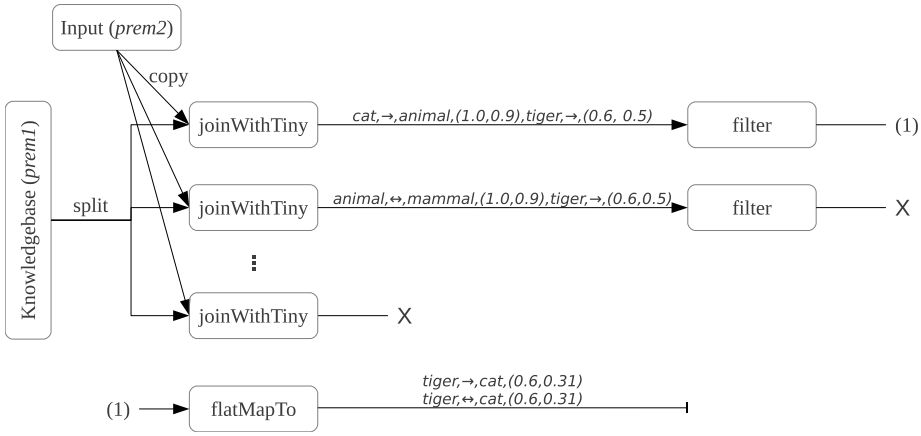



Figure 3.6: Possible execution of the Scalding job presented in Listing 3.7. The entire job can be executed using only mappers.
Slika 3.6: Moguć proces izvršavanja *Scalding* posla iz Listinga 3.7. Ceo posao može biti izvršen koristeći samo mapere.

Listing 3.8: Calculating candidate answers and their expectations and syntactic simplicities for questions in form of in form of *S copula* ? and ? *copula P*.

```
val candidates =
  if (question.subject == AtomicTerm.Question) { // ? copula P
    kb.filter(p, c) { (p: Term, c: Copula) =>
      p == question.predicate && c == question.copula
    }
    .mapTo((s, fc) -> (‘answ, ‘exp, ‘simp)) { (s: Term, fc: Truth) =>
      (s, fc.expectation, 1.0 / s.complexity) // parameter r = 1
    }
  } else { // S copula ?
    kb.filter(s, c) { (s: Term, c: Copula) =>
      s == question.subject && c == question.copula
    }
    .mapTo((p, fc) -> (‘answ, ‘exp, ‘simp)) { (p: Term, fc: Truth) =>
      (p, fc.expectation, 1.0 / p.complexity)
    }
  }
```

Function `sortWithTake` sorts the group of elements and keeps only the first $n$ of them (here: 1), while the final `project` function keeps only the `answer` part of the tuple. Possible execution of this job is presented in Fig. 3.7.

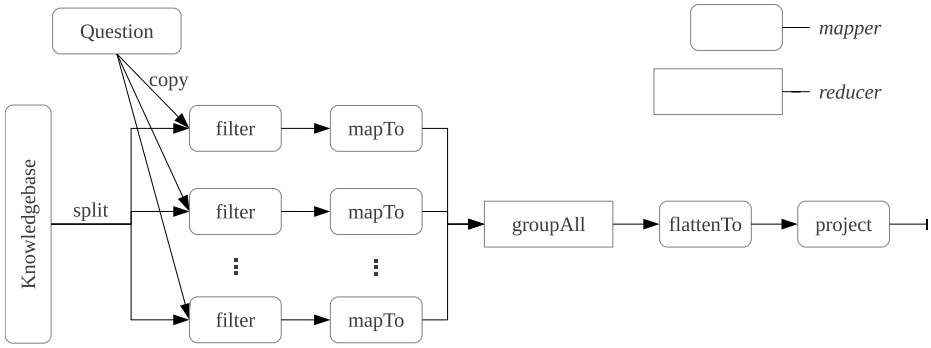



Figure 3.7: Execution scheme of the question answering job presented in Listings 3.8 and 3.9.
Slika 3.7: Šema izvršavanja posla za pronalažanje odgovora na pitanja iz Listinga 3.8 i 3.9.

Listing 3.9: Choosing the best answer from the pipe of candidate answers.

```
candidates
  .groupAll {
    _.sortWithTake((‘answ, ‘exp, ‘simp) -> ‘temp, 1) {
      (a1: (Term, Double, Double), a2: (Term, Double, Double)) =>
        val absDiff = math.abs(a1._2 - a2._2)
        // if the two competing answers have the same e,
        // the simpler answer is chosen
        if (absDiff < Global.EPSILON) a1._3 < a2._3
        // if the expectations differ,
        // the answer with higher e * s is selected.
        else a1._2 * a1._3 > a2._2 * a2._3
    }
  }
  .flattenTo[(Term, Double, Double)](‘temp -> (‘answ, ‘exp, ‘simp))
  .project(‘answ) // keep only the answer
```

Obviously, the realization of DNARS inference engines in form of MapReduce algorithms is not straightforward. One of the main disadvantages is that it does not allow us to directly express our thoughts; by reading the function for abduction, comparison, and analogy in Listing 3.7, for example, it is not immediately obvious how it operates or what it does. The situation becomes worse once other, higher-level NAL rules are introduced. Due to these and other issues discussed earlier, DNARS uses the Titan-based implementation by default.

## 3.8 Summary

DNARS represents a general-purpose reasoning system that combines the powerful NAL formalism with the state-of-the-art algorithms and technologies for large-scale graph processing. Its unique scalable architecture enables it to efficiently manage large knowledge bases, and serve multiple concurrent users, as it will be demonstrated in Section 5.3.

DNARS implements the concepts and inference rules of the first four NAL layers, including:

- First-order inheritance and similarity statements;
- Deduction, induction, and abduction, as well as comparison, analogy, and resemblance;
- Choice and revision; and
- Relational terms.

Although NAL includes more concepts in these first four and, especially, higher layers, the given set of copulas and inference rules is sufficient for performing cognitive tasks with practical applications, as presented in Section 5.4.

DNARS is designed as a standalone system, and provides a common interface that can easily be used by any external client, including human users and software agents. However, its primary purpose is to act as a support for deliberative agents in the Siebog multiagent framework presented in the next chapter.

# Chapter 4

# The Siebog Multiagent Middleware

The HTML5 standard represents one of the driving forces behind the recent proliferation of web applications. With much of the technologies implemented in all major web browsers, HTML5-based application can be executed without any modifications on a wide variety of hardware and software platforms, including standard desktop computers, smartphone and tablet devices, Smart TVs, etc.

Improvements brought by HTML5 on the client side have been matched by corresponding server-side technologies. The *Enterprise Edition* of the Java platform (Java EE) represents one of the most widely used technologies for server-side software development. It offers a wide range of technical solutions suitable for developing scalable, secure, and reliable software systems.

This chapter introduces a novel multiagent middleware, named *Siebog*[1], which includes the following two main components:

- *Extensible Java EE-based Agent Framework* (XJAF), a server-side multiagent middleware with the support for clustered environments (Mitrović et al., 2012a, 2014b; Vidaković et al., 2013).

- *Radigost*, a client-side multiagent platform built on the HTML5 and related standards (Mitrović et al., 2014; Mitrović et al., 2014a).

In addition, Siebog has been integrated with DNARS in order to support agents with advanced reasoning capabilities. The main features of Siebog and advantages over existing multiagent solutions are summarized in the following section.

[1] In the old Slavic mythology, Siebog was a god of love and marriage, which reflects the fact that our Siebog middleware was build by joining two existing systems – XJAF and Radigost. Similarly, Radigost was a god of hospitality. The name roughly translates to *dear guest*, indicating that Radigost agents are dear guests of the client devices.

## 4.1 Main features of Siebog

During the last decade, there has been an obvious paradigm shift in software development. The *web* has evolved into an environment capable of providing functionalities not so long ago available only in desktop applications. And since web-based applications can mostly be executed as native applications on portable devices (Xanthopoulos and Xinogalos, 2013), the desktop-only technologies are becoming less and less important.

Currently, there exists a large number of both open-source and commercial agent middlewares (Bordini et al., 2006; Bădică et al., 2011). As discussed in Chapter 6, however, none of these systems has fully exploited the advantages of web environments. Some efforts towards extending existing systems with web support have been made, but usually in a inefficient manner. For example, in many Java-based middlewares, such as JADE (Bellifemine et al., 2007) or *JaCaWeb* (Minotti et al., 2010), the extensions are based on Java applets. But Java applets requires a browser plug-in, which is unavailable on some platforms (e.g. iOS and Smart TVs). Their applicability is therefore limited to a narrower set of hardware and software configurations.

The goal of Siebog is to provide an infrastructure for executing agents in web environments, but in accordance to the modern standards. It builds on our previous two systems, XJAF and Radigost, in a way that not only combines their individual functionalities, but also results in new features. On the server (Section 4.2) Siebog offers:

- Scalability: agents are automatically distributed across the cluster in order to reduce to computational load of individual nodes. This makes Siebog suitable for applications that need to launch large populations of agents in a computer cluster (e.g. Panigrahi et al., 2011; Simon, 2013).
- Fault-tolerance: the state of each server-side component, including agents themselves, is copied to other nodes. This makes the whole system resilient to hardware and software failures.

The support for distributed execution is present in almost all existing agent middlewares. However, most existing systems use plain computer networks and/or implement their own approaches for agent load-balancing and fault-tolerance (e.g. Alberola et al., 2013; Bellifemine et al., 2007; Faci et al., 2006; Siracuse et al., 2007). One disadvantage of these approaches is lower flexibility. For example, in JADE the agent developer needs to manually specify which agent is hosted on which computer, while in Siebog this process is performed automatically.

With Siebog we also demonstrate that it is not necessary to "reinvent the wheel." Instead, it is more beneficial to use existing, standards-compliant, and well-tested solutions offered by Java EE. This approach also increases the interoperability of Siebog, since its agents can seamlessly be integrated into non-agent enterprise software. This in turn may help in bridging the gap between the agent technology and the industrial software systems.

The client-side component of Siebog (Section 4.3) has the following set of unique characteristics:

- It is platform-independent, and supports a range of hardware and software platforms (Mitrović et al., 2014; Mitrović et al., 2014a). To agent developers, this provides the *write once, run anywhere* approach. The end-users, on the other hand, can utilize the benefits of the agent technology in the most convenient manner.
- It requires no prior installation or configuration steps.
- Its client-side runtime performance is comparable to that of a classical, desktop multiagent platform (Mitrović et al., 2014a).

Currently, there exists only one fully-featured HTML5-based middleware (Jarvenpaa et al., 2013). Although it conveniently relies on a JavaScript-based server, it does not use the full set of HTML5-related standards, such as Web Workers and WebSockets, and also lacks Siebog's server-side features.

It is worth noting that Siebog is more than just a sum of its individual components. As presented in Section 4.4, Siebog also features:

- Cross-platform messaging: client-side agents can communicate with server-side agents or even client-side agents hosted in different browsers/devices. The communication is performed transparently, i.e. as if all agents are located at the same place.
- Code sharing: an agent written once can be executed both on the client and on the server.
- Heterogeneous agent mobility: an agent can move freely between the client and the server.

Finally, by using DNARS as the intelligence sub-system, Siebog "breaks away" from the traditional BDI model and enables the development of agents with unique reasoning capabilities.

The described range of functionalities is not available in any existing multiagent middleware (Chapter 6). Therefore, Siebog represents a novel approach for deploying intelligent agents in web and enterprise environments.

## 4.2 The server-side architecture

The server-side of Siebog (its XJAF component) is designed as a set of loosely-coupled components called *managers* (Mitrović et al., 2012a, 2013b; Vidaković et al., 2013). Each manager is dedicated to handling a distinct part of the overall functionality. A manager is represented and used only by its interface, and even multiple implementations of the same interface can be active simultaneously.

The manager-based approach offers the highest level of flexibility, and allows third-party re-implementations of individual components. The three most

important managers are *Agent Manager*, which controls the agent life-cycle, *Message Manager*, which handles the inter-agent communication, and *Connection Manager* in charge of maintaining networks of distributed Siebog clusters.

The first version of XJAF was described in 2002 (Vidaković and Konjović, 2002) and has since been revised several times. The latest incarnation – the one used in Siebog and described in this section – is focused on harnessing previously described benefits of computer clusters: scalability and fault-tolerance. It achieves these features by relying on a range of Java EE technologies (Vidaković et al., 2013), the most important of which are *Enterprise JavaBeans*.

### 4.2.1 An overview of Enterprise JavaBeans

*Enterprise JavaBeans* (*EJB*s, or simply *beans*) implement the business logic of an enterprise application (DeMichiel and Keith, 2006). EJBs are often described as *managed* components, since their life-cycle and behavior is controlled by an *enterprise application server*.

In general, there are two categories of EJBs (DeMichiel and Keith, 2006; Mitrović et al., 2012b, 2013a): *message-driven*, and *session*. A message-driven bean is used along with the *Java Message Service* (JMS)[2], an additional Java EE technology that deals with asynchronous messaging. In the context of JMS, message-driven beans act as message receivers: they are never invoked directly but are instead used to process messages in the JMS pipeline.

Session beans can further be categorized into *singleton*, *stateless*, and *stateful*. As its name suggests, there is a single instance of a singleton EJB per Java Virtual Machine (VM) instance. Concurrent access is managed by the *EJB container* and can be fine-tuned by the developer. Stateless EJBs maintain no conversational state between distinct invocations. They are well-suited for operations that can be executed in a single method call. A stateful EJB, on the other hand, is used when the client requires an ongoing, more complex conversation.

Stateless EJBs offer the best runtime performance, since the application server does not have to maintain the conversational state. When a request arrives the server can freely create a new stateless EJB instance on any node in the computer cluster. However, they have a limited application in the context of agent development since two messages cannot be delivered to the same EJB. More details about the concrete uses of EJBs on the Siebog server and their benefits are given in the remainder of this section.

[2] `http://www.oracle.com/technetwork/java/index-jsp-142945.html`, retrieved on August 12, 2014.

### 4.2.2 Agent management

A server-side Siebog agent can be represented by a stateless or a stateful EJB. The choice of which concrete category to use has an implication on the agent's runtime behavior, as discussed later.

It has been argued that EJBs, as reactive components, might not be suitable for developing more complex agent architectures (Luck et al., 2004). While in their simpler form, Siebog agents do operate by reacting to external messages, the system includes a service that can be used to implement more complex behavior. The idea is to register an internal timer (*heartbeat*) to ping the agent at certain time intervals, allowing it to perform tasks when there is no external stimuli. This approach of having an external component that calls predefined methods of the agent object is also found in other software systems for developing reasoning agents, such as Jason (Bordini et al., 2007).

Each server-side agent has its own thread of execution, but there is no thread-to-agent mapping. Instead, the underlying enterprise application server maintains a thread pool and automatically assigns threads to agents as needed. For example, when a message is received, the agent will be given a thread to process it. In the worst-case scenario, when all agents are actively executing tasks, there will be as many threads as there are agents. However, the server will try to reduce the resource consumption when possible by, for example, deallocating threads that have not been used for a certain amount of time. Additionally, if an agent is inactive for a sufficient amount of time, it will be passivated (Goncalves, 2010): removed from the runtime memory and stored on a secondary storage (e.g. the hard-disk). When needed the agent will be re-activated to resume its execution.

The directory of agents is implemented through *Java Naming and Directory Interface* (JNDI)[3], which also works in clustered environments. It enables any interested third-party to find details about available agents with the support for pattern-based searches.

### 4.2.3 Clustering features

The organization of an Siebog cluster is shown in Fig. 4.1. A single node within the cluster is described as *master*, while the others (zero or more) are described as *slaves*. Within a node, the *host controller* is used to manage the Siebog instance (Marchioni, 2014). In addition, the master node can be used to remotely control the entire cluster, through the *domain controller* (Marchioni, 2014). This is the only difference between the master and the slaves; all nodes in a cluster have the same execution priority, can directly communicate to each other, etc.

Siebog managers are designed to be completely independent of each other. They share information with each other through a distributed, concurrent, and highly-efficient *Infinispan* cache (Marchioni and Surtani, 2012). Whenever it runs a new agent, for example, the Agent Manager stores all the necessary

[3] `http://www.oracle.com/technetwork/java/jndi/index.html`, retrieved on August 12, 2014.

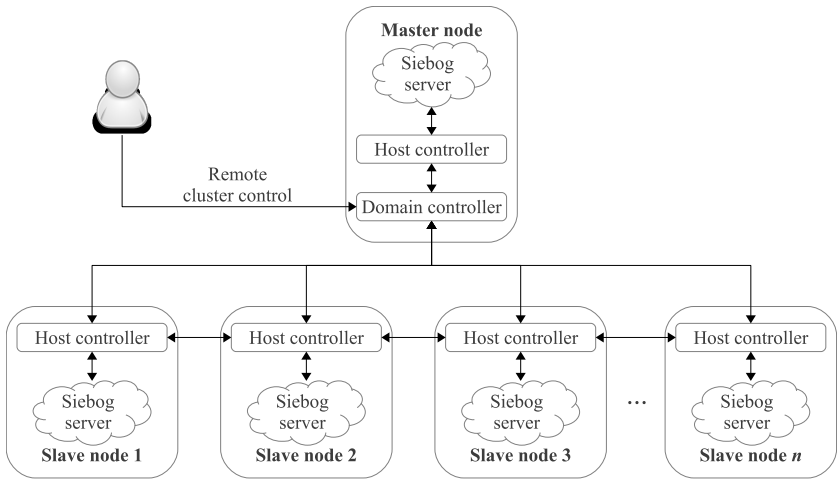



Figure 4.1: Siebog operates in a symmetric cluster: each node is connected to every other node. A single node is recognized as the *master* and can be used to remotely control the cluster.
Slika 4.1: Siebog funkcioniše u simetričnom klasteru: svaki čvor je povezan sa svakim drugim čvorom. Jedan čvor, označen kao *gospodar*, se može koristiti za udaljenu kontrolu klastera.

information in the cache. This information can later be retrieved by the Message Manager to deliver a message to the agent. Since the cache is distributed across the cluster, the managers themselves can be hosted on any node.

The cluster has two main functionalities: *state replication and failover* and *load-balancing*. State replication and failover are applicable to stateful EJBs only. Whenever a stateful EJB's internal state is changed, the replication process copies it across other nodes in the cluster. In case the EJB's node becomes unavailable, the failover process fully restores the EJB object on one of the remaining nodes. From the client's point of view, the entire process is executed transparently: all subsequent method invocation will end-up in the newly created object.

Two state replication modes are supported (Surtani et al.): *replicated* and *distribution*. The replicated mode copies the state across all nodes in the cluster. It can withstand high failure rates, but works efficiently only in clusters that consists of up to 10 nodes (Surtani et al.). The distribution mode, on the other hand, is more suitable for larger clusters, as it copies the state to a configurable number of nodes. It uses hashing algorithms and parallel execution to achieve linear scaling as more nodes are added to the cluster. The distribution mode includes other advanced features as well, such as L1 caching (Surtani et al.).

Load-balancing is used to automatically distribute agents across different nodes in the cluster, and to speed up the overall runtime performance of server-side Siebog. It works with both stateful and stateless EJBs, although the be-

havior is slightly different. When the client creates a new stateful EJB instance, the server places it in one of the available nodes, and all subsequent invocations of the EJB's methods end-up there. In case of stateless EJBs the load-balancing works on a per-method basis. When the client makes a request to a stateless EJB, a new instance is created on one cluster node. Once it serves the request, the instance is destroyed.

The described load-balancing process has a major implication on the development of agents. If an agent is based on a stateless EJB, it becomes theoretically impossible to send it more than one message. Since there is no state sharing between distributed stateless EJBs, two consecutive messages sent to the agent might end-up in two unrelated EJB instances. Even a seemingly simple operation, such as replying to the message sender, cannot be performed. Given these properties, as well as the lack of state replication and failover, stateless EJBs have only a limited application in Siebog; in the majority of cases, stateful EJBs should be used.

### 4.2.4 Message management

Siebog agents, both on the server and on the client side, communicate by exchanging messages based on the standard FIPA ACL (FIPA Acl). The exchange is asynchronous, although a number of methods is provided to enable blocking behavior, in order to simplify agent development in certain scenarios.

Java EE includes a communication architecture named *Java Message Service*[4] for asynchronous message exchange between loosely-coupled components. It supports two communication patterns: *point-to-point*, and *publish-subscribe*. In the first pattern, a *producer* places messages in a *queue* to be processed by a single *consumer*. The publish-subscribe pattern is realized around a *topic*: the producer sends a message to the topic to be processed by all subscribed consumers.

In the server-side Siebog, the communication is achieved via the point-to-point model with the Message Manager acting as the producer. Messages are published to a queue and consumed by message-driven beans (MDBs). MDBs are organized in a pool, which can automatically grow (and shrink) according to the demand.

MDBs deliver messages to target agents by invoking the appropriate method of the agent class. Once successfully processed, the message is acknowledged and then removed from the queue. If, however, there is an error, the message is requeued and the delivery is retried at a later time. After a number of unsuccessful deliveries, the message is stored in the so-called *dead-letter queue* and can be inspected and processed manually. All of these and other issues (such as message ordering, concurrent access, cluster-wide coordination, etc.) are handled by the underlying JMS.

[4] `http://www.oracle.com/technetwork/java/index-jsp-142945.html`, retrieved on August 12, 2014.

## 4.3 The client-side architecture

The client-side of Siebog (its Radigost component) is designed to support the execution of agents in web environments. These include web browsers but any JavaScript runtime can be used[5]. On the client, Siebog provides the necessary infrastructure for the deployment, execution, and interaction of agents. Its core functionalities include agent life-cycle management, a communication infrastructure, and a yellow-pages service. Additionally, it supports agent state persistence, which allows an agent to become detached from its host web page, and run across multiple browser sessions.

Siebog depends on a range of HTML5 and related standards for its client-side functionalities, the two most important of which are *Web Workers* and the *WebSocket* protocol. These two technologies and their respective roles in Siebog are described next.

### 4.3.1 Mapping agents to Web Workers

Web Workers define a model for true multi-threading and asynchronous messaging in JavaScript[6]. Before the introduction of Web Workers, all JavaScript applications were inherently single-threaded, with developers having to rely on timers and scheduling in order to simulate asynchronous code execution.

Web Workers are based on the *actors* concurrency model initially proposed in (Hewitt et al., 1973). Unlike classical threads actors utilize a *share-nothing* approach: an actor shares none of its runtime resources with other actors. It operates as a self-contained entity, whose only means of communication with the environment and other actors is message exchange. The exchange of messages might be slower than the shared memory, but it relieves the developer from having to worry about proper synchronization techniques – the advantage especially important in complex software systems.

A software agent and an actor share several core properties: both are self-contained entities, with their own threads of execution and relying on messages to communicate with other entities. Therefore, the mapping from agents to actors/Web Workers is a natural process. A multiagent platform could be implemented in JavaScript without using Web Workers, but the development process would become more difficult because of the need to efficiently simulate multiple threads. This would undoubtedly affect the execution performance as well. The Web Workers technology and modern web browsers also provide an efficient communication infrastructure (e.g. message queuing and ordering). Having all these benefits in mind, client-side Siebog agents are realized in form of Web Workers.

[5]Such as *Node.js*, `http://nodejs.org`, retrieved on August 12, 2014.
[6]`http://www.w3.org/TR/workers/`, retrieved on August 12, 2014.

### 4.3.2 Two-way communication through WebSockets

For an extended period of time, the client-server communication on the web could be performed using the *polling* model only. In the polling model, when the client needs some information from the server, it must initiate a request and receive the required information in response. The *AJAX* technology (*Asynchronous JavaScript and XML*) has improved the polling model significantly by enabling asynchronous communication and eliminating the need to refresh the entire page only to display the newly retrieved information. Although the polling model was sufficient for many web applications, there was still a need for a long-held connection between a web client and a server.

The *WebSocket* protocol represent relatively new, but standardized technology for establishing persistent TCP connections between web clients and servers[7]. It enables *full-duplex* connections, i.e. simultaneous, two-way exchange of text and binary information over the same channel. It brings the *push* model to modern web applications: once the connection is established, the server can initiate information delivery on its own, and even make requests to the client.

The WebSocket protocol is designed to be backwards-compatible. The initial request for establishing a persistent connection is incorporated into the *HTTP Upgrade* header. The WebSocket-enabled server recognizes this request, sends an agreement to the client, and then establishes the full-duplex communication over the existing TCP connection. Additionally, the protocol uses only ports 80 (HTTP) or 443 (secure HTTP – HTTPS), requiring no configuration changes to the server's firewall.

The use of WebSockets is very important for the practical application of Siebog. It enables a third-party, server-side agent to take the initiative and start the conversation with a client-side agent. Without WebSockets it would be possible only for client-side agents to send requests to the server, and not the other way around, which would severely limit the flexibility of our system.

### 4.3.3 Client-side agents

In order to facilitate the development and execution of client-side agents, Siebog provides a dedicated *client library*. At its core, the library includes a number of agent *prototypes*. The basic prototype (*Agent*) defines functionalities common to all agents. Other prototypes are included as well; for example, *CNetContractor* can be used to model the role of a *Contractor* in the standard *Contract Net* protocol (FIPA CNet). As a simple example of using prototypes, Listing 4.1 shows a *Calculator* agent based on the *Agent* prototype, which simply returns the sum of two received numbers.

As shown, communication between client-side Siebog agents is also based on the FIPA ACL format. The basic representation of a FIPA ACL message is also included in the client library and it is shown in Listing 4.2. As noted in (FIPA Acl), the FIPA specification defines many more message fields than shown here. However, since JavaScript allows the addition of new object properties at

[7] `https://www.websocket.org/`, retrieved on August 12, 2014.

runtime, only the minimally required set of fields is included by default in order to optimize the runtime performance.

Listing 4.1: A simple example of a Calculator client-side Siebog agent.

```
importScripts("/siebog/radigost.js"); // import the client library

// defining a new agent based on the Agent prototype
function Calculator() { };
Calculator.prototype = new Agent();

Calculator.prototype.onMessage = function(msg) {
  var sum = msg.a + msg.b;
  // ACLMakeReply is a helper function for constructing a reply to 'msg'
  var reply = ACLMakeReply(msg, Performative.INFORM, this.aid);
  this.post(reply);
};
```

Listing 4.2: Definition of a FIPA ACL message used on the client side of Siebog.

```
function ACLMessage(performative) {
  // a constant in ACLPerformative object
  this.performative = performative;
  // an array of receivers
  this.receivers = [];
  // message content
  this.content = null; };
```

In addition to inter-agent communication, there needs to be a way for an agent to communicate to its host environment (e.g. the web page). This communication is realized through the *Observer* software design pattern (Purdy and Richter). The client library includes a prototype named *AgentListener* with the following set of methods:

- *onStart(aid)*: The agent has been started. The parameter represents the agent's identifier.
- *onStop(aid)*: The agent has been stopped.
- *onError(aid, msg)*: An error has occurred while running this agent. The optional *msg* parameter might provide more details about the error.
- *onStep(aid, msg)*: The agent has finished a "computational step". The optional *msg* parameter might include more information, such as the computational sub-result.

The first three methods are invoked by the client library, while the last one is optional and can be invoked by the agent itself.

Each client-side agent is assigned a globally-unique *agent identifier* (AID), which is also used in the aforementioned functions. An AID is a string in the

form of `name@hap#user`. Here, `name` is locally-unique name of the agent, used to differentiate between, for example, agents *Smith* and *Jones* hosted in the same web page. The `hap` part is globally-unique domain name of the Siebog application, such as *example.org*.

The `user` part is used to identify and extend the user's session and his/her interaction with the same agent. If a web application is available to registered users only, then the `user` part of an AID can be some server-generated key tied to the particular user. In a web-based learning system this would be a unique string that identifies the student. For general-purpose applications that do not require users to register, the `user` part of the AID can be generated in many different ways, e.g. from the HTTP session. *Cookies*, or more modern HTML5 web storage options[8] can be used to preserve the session ID locally.

### 4.3.4 Agent state persistence

The lifespan of a client-side agent is inherently tied to its host web page. Once the user loads another web page or closes the browser, the agent is automatically destroyed. However, for any meaningful application of Siebog it is crucially important for the client to be able to continuously interact with the same agent over an extended period of time and across multiple browser sessions. For example, in an online learning and tutoring system, the pedagogical agent needs to be able to track student's progress through a course during the entire semester. To achieve this functionality, Siebog relies on the concept of persistent agent state.

On the server agent state is stored in form of a *key-value* pair, where AID is used as the key. The actual state can be practically any JSON (*JavaScript Object Notation*[9]) object. When the user visits a web page that hosts Siebog agents, any previously stored states are automatically downloaded and injected into agents. Similarly, as the user leaves the page, agent states are automatically stored on the server. The base *Agent* prototype defines two functions, `getState()` and `setState(state)` for, respectively, retrieving and injecting the agent state. If an agent does not require state persistence these two functions can simply be left unimplemented.

The agent state persistence sub-system provides the possibility for implementing agent mobility. In the context of Siebog, a mobile agent can migrate from the web browser of one client to the web browser of another, and continue its execution there. During the migration process, the agent stores its state on the server, and then the server pushes it to the target client over the WebSocket protocol. An example of a mobile agent and its usage is shown in Section 5.1.

Once the HTML5 web storage specification has been standardized and fully supported by modern web browsers, Siebog will use it as a temporary storage. This will help to reduce the server bandwidth when the user is simply leaving the current page to load another page of the same Siebog web application. For now, other approaches can be employed (e.g. the HTML *iframe* element).

[8] `http://dev.w3.org/html5/webstorage/`, retrieved on August 12, 2014.
[9] `http://json.org`, retrieved on August 12, 2014.

### 4.3.5 Security concerns

The popularity and wide-spread usage of JavaScript have resulted in the language being increasingly used as a tool for malicious browser-based attacks (Guarnieri and Livshits, 2009; Yu et al., 2007). Because of the issues, JavaScript code is often subjected to a range of limitations implemented at the browser level. For example, JavaScript applications are executed in a *sandbox* and are forbidden any direct access to system resources, such as the file system. Additionally, a JavaScript application can only access remote resources hosted on the same domain as the script itself – a limitation known as *Same Origin Policy*[10]. These are only some of the limitations any Siebog developer needs to be aware.

An important thing to note is that JavaScript is an interpreted language: the source code of a JavaScript program is sent to and executed by the web browser, making it open to any interested party. Although there exist various source code obfuscation tools (e.g. YUI[11]), client-side agents should not include any proprietary algorithms. One possible solution for this problem would be to leave the proprietary algorithms in a server-side agent, and then interact with them as described in the next section.

When it comes to the server-to-client communication through WebSockets, all server-side components are under the direct control of the system administrator. This can prevent an unwanted deployment of any malicious code. To increase the user's trust in a Siebog-based application administrator can install security certificates. For the confidentiality of exchanged information, they can additional employ WebSockets' inherent support for secure HTTP connections (i.e. HTTPS).

## 4.4 Client-server integration

Heterogeneous system integration is a common and a well-understood problem. Several integration patterns have emerged over time: the *Shared Database*, *Message-Oriented Middleware*, and *Remote Procedure Invocation* pattern (Hohpe and Woolf, 2003). Shared Database is applicable when different sub-systems need to share the data but otherwise operate independently of each other. The database is directly accessible by all components and usually provides a single schema. The Message-Oriented Middleware pattern offers the greatest degree of component independence (Hohpe and Woolf, 2003). Different parts of the system exchange messages, carrying (usually) small packets of information in an asynchronous manner.

Remote Procedure Invocation enables heterogeneous sub-systems of the overall application to share their functionalities rather than data (Hohpe and Woolf, 2003). The public functionality of each sub-system is exposed using an agreed-upon format. During the invocation, all internal communication (i.e. within a

[10] `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`, retrieved on August 12, 2014.

[11] `http://yui.github.io/yuicompressor/`, retrieved on August 12, 2014.

sub-system) is automatically transformed into a standardized external protocol. Over time, Remote Procedure Invocation has been realized in a number of concrete forms, including CORBA, Java RMI, and web services, where web services currently represent the most widely-used approach.

In case of Siebog, the integration approach is mainly dictated by their underlying implementation technologies. The most natural and straightforward way of integrating the JavaScript-based client and the Java EE-based server is to use the Remote Procedure Invocation pattern, and its realization based on web services.

### 4.4.1 A web services-based layer

The goal of web services is "to support interoperable machine-to-machine interaction over a network."[12]. In general, a web service consists of an interface understandable by machines (and humans), and a communication protocol. The first step in developing Siebog is, therefore, to provide web service-based interfaces for its managers. This, however, can be achieved in different ways.

*XML web services* represent one of the two competing approaches for developing and using web services. It encompasses a wide range of standards and specifications[13], covering interface definition, description and discovery, communication, security, etc. Communication is in most cases performed using XML-encoded messages transmitted over HTTP, although other approaches are possible as well. Unfortunately, the sheer amount of (sometimes conflicting) standards and specifications related to XML web services has turned out to be their weakest point preventing them from gaining much traction.

*Representational state transfer* (REST) is a more recent, alternative design approach for applications based on web services (Fielding, 2000). It uses the four HTTP operations – GET, POST, PUT, and DELETE – to query and manipulate resources. Resources themselves are represented using *Uniform Resource Identifiers* (URIs). REST is a "standard-less" set of architectural design principles and constraints. The *Stateless* constraint, for example, states that all communication between the client and the server is stateless, in the sense that the server should not store any contextual information about the client (Fielding, 2000). Web services that adhere to all of REST principles and constraints are often referred-to as *RESTful*.

It is worth noting that an older version of XJAF has been also provided in form of a web service-oriented architecture, with its managers designed as XML web services (Mitrović et al., 2012a). However, RESTful web services represent a "better fit" for the intended purpose of integrating JavaScript and Java EE systems. They are much easier to use from the JavaScript client, especially when JSON format is used to represent objects. In addition, RESTful services provide better performance due to less runtime overhead (e.g. Mulligan and Gracanin, 2009).

[12] `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice`, retrieved on August 12, 2014.

[13] `http://www.w3.org/2002/ws`, retrieved on August 12, 2014.

Table 4.1: A subset of the Agent Manager's REST API. All methods consume and produce objects of type `application/json`. Parts of URIs enclosed in curly braces represent variables.
Tabela 4.1: Podskup *REST API*-a Agentskog menadžera. Svi metodi prihvataju i vraćaju objekte tipa `application/json`. Delovi *URI* adresa u vitičastim zagradama predstavljaju promenljive.

| Method | URI | Description |
|---|---|---|
| GET | /classes | Returns the list of available agent classes. |
| GET | /running | Returns the list of running agents (their AIDs). |
| PUT | /running/{agClass}/{name} | Runs a new agent of the given class, and with the given runtime name. |
| DELETE | /running/{aid} | Stops the given agent. |

Since EJBs are used to implement major parts of the server-side Siebog, including the managers, the process of transforming them to RESTful web services is straightforward. This is one example of how the standards-compliance can bring benefits to software development. The majority of work has consisted on annotating the appropriate parts of code. Custom (de-)serializations for JSON messages had to be provided in some cases (e.g. for objects representing FIPA ACL messages), but the entire process was completed without any technical difficulties.

Table 4.1 outlines the proposed REST interface of the Agent Manager. Its base URI is "/agents", and in all cases the input arguments and return values are represented as JSON-formatted strings.

Although managers have been re-designed as RESTful web services, internal Java components, such as agents, still invoke them as regular EJBs. This is because EJB invocations incur far less overhead than REST interfaces. For example, when both the agent and the manager are located on the same machine (which is the usual case), no serialization of method parameters is required. Luckily, REST interface definitions can be mixed in with regular EJB method implementations. Listing 4.3 shows the header of the Message Manager's `post` method, which is used to send a FIPA ACL message.

The given method can be invoked both through the REST API and using standard EJB invocation approach. The client-side agents can use it to send messages to server-side agents, or even to other client-side agents hosted in different web pages and different devices. This is achieved by providing client-side agents with *stub* implementations of server-side managers. An example stub implementation for the Message Manager's `post` method is given in Listing 4.4.

Listing 4.3: Header of the Message Manager's `post` method, used for sending FIPA ACL messages. The method can be invoked both using Java Remote Invocation and REST

```
@POST
@Path("/")
@Consumes(APPLICATION_FORM_URLENCODED)
@Produces(APPLICATION_JSON)
@Override
public void post(@Form ACLMessage msg) { ... }
```

Listing 4.4: Client-side stub implementation of the Message Manager's `post` method.

```
XJAF.post = function(msg, onSuccess, onError) {
  $.ajax(XJAF.messageManager, {
    type : "POST",
    data : JSON.stringify(msg),
    success : onSuccess,
    error : onError
  });
};
```

Once the web service layer is added on the server, the two sides of Siebog can be integrated into a single system. This final integrated architecture of Siebog is outlined in Fig. 4.2. As shown, in addition to stub implementations of manager on the client, each client-side agent has its own stub counterpart on the server. To external entities, a stub appears as a regular server-side agent, but any messages sent to it will be forwarded to the concrete client-side agent.

The use of stubs does not introduce more computational overhead than necessary. In the client-server agent communication, messages have to be transferred as JSON strings. Instead of having a centralized repository of client-side agents which also acts as a message (de-)serializer, a more efficient approach is used (e.g. no bottlenecks, no single point of failure, etc.). Since the agent identifier on the client includes the browser session identifier, it is impossible for two different client-side agents to reference the same server-side stub.

In order to facilitate server-to-client messaging, a new manager (*WebClient*) is introduced. The new manager provides two main functionalities. First, it acts as a WebSocket server endpoint, keeping the track of all active client-side, and forwarding messages from server-side components (incl. agents). Secondly, the new manager is in charge of agent state persistence described earlier.

The integrated Siebog architecture enables transparent inter-agent communication and action coordination, regardless of the types and physical locations of agents. A client-side agent agent can send a message to a server-side agent via the appropriate stub call. However, if the target agent is actually a stub representation of a different client-side agent, the message may end up in a different web page or on a different device. This opens up a range of possible

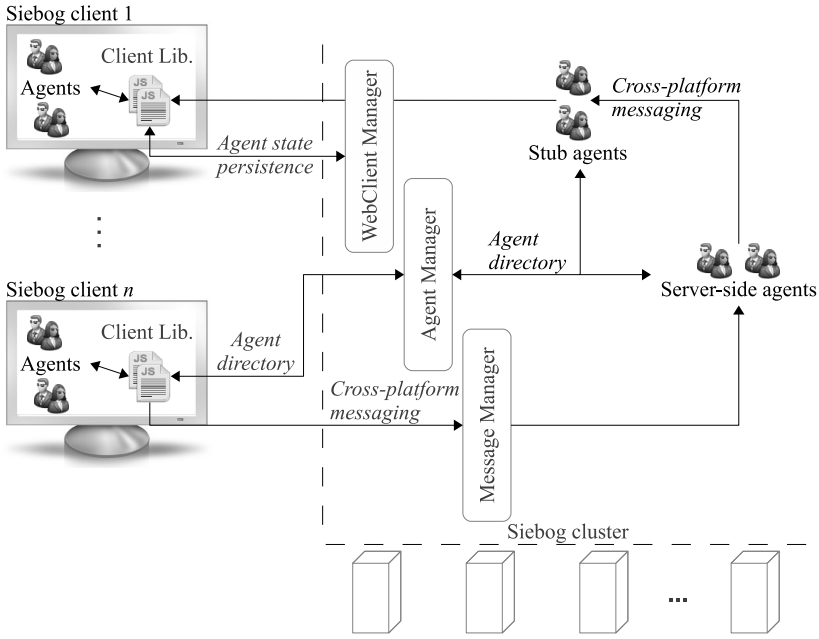



Figure 4.2: The final integrated architecture of Siebog.
Slika 4.2: Konačna integrisana arhitektura Siebog-a.

practical applications; for example, in case of *smart environments* (Nakashima et al., 2010) agents hosted in physically distributed smart objects can seamlessly exchange information and coordinate their actions.

### 4.4.2 Cross-platform interaction

In Siebog, both client-side and server-side agents can initiate an interaction. As shown in Fig. 4.2, client-side agents use stub implementations of server-side managers to perform asynchronous AJAX calls to the appropriate RESTful services. On the other hand, when a server-side agent needs to interact with a client-side agent, the standard WebSocket protocol is used. The message is serialized on the server side into a JSON-formatted string, transferred to the client's web browser, de-serialized into a corresponding message object, and delivered to the target. Unfortunately, the (de-)serialization process cannot be fully avoided at the moment, as web browsers in general do not support binary data transfer through WebSockets. Fig. 4.3 shows an example of a client-side agent that creates, interacts with, and finally destroys a server-side agent.

As noted earlier, the interaction between client and server side is manifested in three different ways: code sharing, message exchange, and agent mobility. Obviously, code sharing is possible as long as the agent implementation satisfies all the constraints imposed by web environments and relies only on libraries

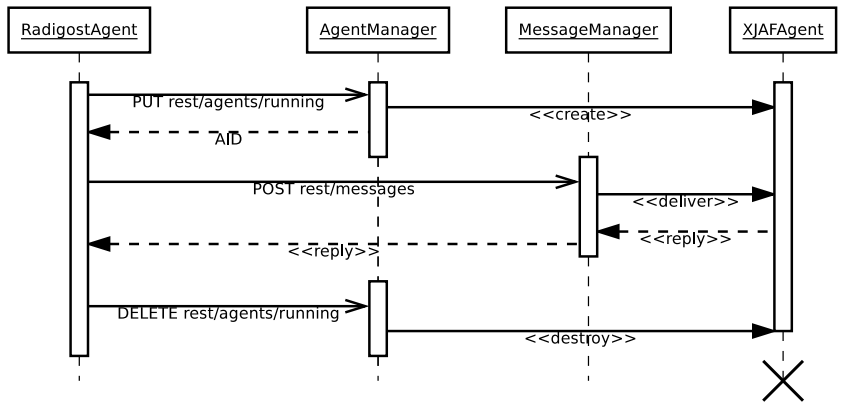



Figure 4.3: Sequence of messages initiated by a client-side agent which starts, interacts with, and finally destroys a server-side agent.
Slika 4.3: Sekvenca poruka koju je inicirao klijentski agent, a koji pokreće, komunicira sa, i konačno uništava serverskog agenta.

available in both JavaScript and Java.

Message exchange is the easiest to achieve. It is enough to extend the agent identifier (AID) representation with a *platform identifier*. The message sending routines in both the server and the client side can then simply compare this value with their host platforms' identifiers, and forward the message appropriately.

The idea of code sharing is that the agent developer can write an agent once, using his/hers preferred programming language. The Siebog platform then takes care of executing the agent on the client and on the server, as needed. This feature has two aspects: executing JavaScript agents in the Java VM, and executing Java agents in web browsers.

The execution of JavaScript agents on Java VM is a much simpler task. *Java Specification Request* (JSR) 223 defines the standard *Scripting API* for Java VM (Grogan, 2006). Besides executing JavaScript code, the API offers some advanced features. For example, JavaScript programs can import and use Java classes, indirectly implement Java interfaces, which are then directly accessible in Java programs, etc.

Unfortunately, there is no standard way of executing Java code in web browsers. The approach of embedding server-side agents into Java applets would be *subpar*. However, an efficient third-party solution does exist. *Google Web Toolkit* (GWT) is a popular set of open-source libraries and tools that transform complex Java-based web applications into pure JavaScript applications (Tacy et al., 2014). One of its defining features is cross-browser compatibility: GWT will produce a number of compilations from the same Java source, each optimized for a distinct web browser. In this way developers are relieved from worrying about browser-specific implementations, and the best possible runtime performance can be achieved. Given its many advantages, Siebog relies on GWT for executing Java agents on the client.

Although the code sharing feature of Siebog does work in practice, developers need to be aware of its limitations. For example, writing performance-centric agents in JavaScript and then executing them in Java VM might not be the best option. Instead, it would be better to move the core implementation to a server-side agent, and then communicate with it from the client. Similarly, although powerful, GWT poses a number of limitations on Java implementations; more details are available in the official GWT documentation[14].

The final aspect of the client-server interaction is agent mobility. For example, an agent running in the web browser should be able to migrate to the server, replicate and distribute itself across the cluster, and finally return to the web browser carrying the computational result. With the existence of state persistence and code sharing, this functionality can be achieved easily. An example of its use is shown in Chapter 5.

## 4.5 DNARS-based intelligent agents in Siebog

DNARS was built from ground-up to support a range of external clients. Its entire functionality is exposed in form of a REST API, for example:

- *GET /dnars/anwser?q={question}&domains={domains}*: Returns an answer to *question* while consulting the specified list of *domains*.
- *POST /dnars/judgements?domains={domains}*: Adds new *judgements* to the given list of *domains*. The actual judgements are specified in the request body. This action can trigger the Forward inference engine, which might derive and include even more new judgements.

The architecture of DNARS has previously been presented in Chapter 3 and shown in Fig. 3.1, but it will be briefly summarized here. DNARS consists of two main parts: the Inference engine and the Backend knowledge base. The Inference engine itself includes:

- Resolution engine for answering questions;
- Forward inference engine for deriving new knowledge;
- Short-term memory for storing relevant statements; and
- Event stack for receiving the knowledge base notifications.

The Backend knowledge base is divided into distinct Knowledge domains; domains are (not necessarily disjoint) subsets of the overall knowledge. Finally, changes in the Backend knowledge base are detected and appropriately handled by the Event manager.

Although Siebog agents could simply use the REST API as well, a tighter integration with DNARS has been realized. The two main reasons for doing

[14] `http://www.gwtproject.orc/doc/latest/DevGuide.html`, retrieved on August 12, 2014.

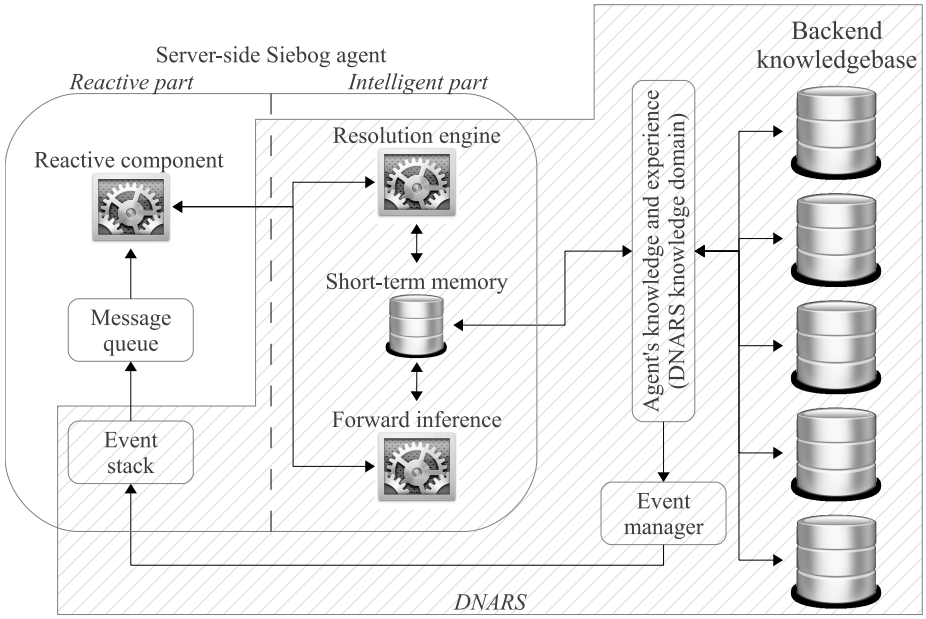



Figure 4.4: The architecture of server-side Siebog agents.
Slika 4.4: Arhitektura serverskih agenata u Siebog-u.

this are lower communication overhead, and a simpler agent development process through agent-oriented programming abstractions. Fig. 4.4 outlines how the building blocks of DNARS have been merged into server-side Siebog agents. As shown, each server-side Siebog agent now consists of the *reactive* and the *intelligent* part. The reactive part simply operates on the basis of external events, such as messages received from other agents, internal triggers, notifications from the knowledge base, etc.

The intelligent part is the DNARS Inference engine. That is, Resolution engine, Forward inference engine, Short-term memory, and Event stack are integral parts of Siebog server-side agents. The Backend knowledge base, on the other hand, is essentially external to the agent. Since the knowledge base can store very large amounts of information, it would be impractical to incorporate it into the agent itself. Finally, the agent's individual knowledge and experience is mapped to a DNARS Knowledge domain.

The final step in Siebog-DNARS integration is to introduce agent-oriented programming abstractions. Their purpose is to enable developers to define agents in the usual sense, e.g. in terms of beliefs and goals.

Since annotations are the standard meta-programming constructs of Java EE, the same approach can be applied for defining server-side DNARS agents. Annotation based development of intelligent agents has also been used elsewhere (e.g. Pokahr et al., 2014). Currently, Siebog agents that rely on DNARS reasoning are programmed in terms of *beliefs* and *actions*. The three main annotations for be-

lief and action management are `@Beliefs`, `@BeliefAdded`, and `@BeliefUpdated`.

As noted in Section 3.6, the Event manager in DNARS features several Event Dispatch Threads (EDTs) that collect events generated in the knowledge base and dispatch them to registered observers. In case of Siebog, these event notifications are dispatched in form of FIPA ACL messages. The Siebog framework uses the *Reflection API*[15] to scan each DNARS agent for annotated methods. It then intercepts all messages delivered to these agents from EDTs, extracts their contents, and calls the corresponding methods. Therefore, the entire process is performed transparently from the agent's and the agent developer's point of view. An example usage is shown in Listing 4.5.

Listing 4.5: Annotation-based belief management in Siebog. Instead of `String`s, the appropriate data types can be used (e.g. `siebog.dnars.base.Statement` for representing statements).

```
public class DNarsExample extends DNarsAgent {
  @Beliefs
  public String[] initBeliefs() {
    return new String[] { "carnivore -> agent_eater (1.0, 0.9)",
      "tiger -> carnivore (1.0, 0.9)"
    };
  }

  @BeliefUpdated(pattern=".")
  public void beliefUpdated(Statement st, Truth oldTruth) { /* ... */ }

  @BeliefAdded(subj="tiger", copula="->", pred="agent_eater", truth=".")
  public void runFromTheTiger(Statement... added) { /* ... */ }
}
```

The given agent has two initial beliefs, defined in the annotated `initBeliefs` method. The agent is notified when new beliefs are added, or when its existing beliefs are updated. The corresponding two annotations support statement filtering. In the given example, the agent will be notified when a new belief $tiger \rightarrow agent\_eater$ is added. It will also be notified whenever any of its beliefs is updated.

The `@BeliefAdded` and `@BeliefUpdated` annotations can be used to define actions carried out by the agent. The method `runFromTheTiger` can be seen as an action triggered when the agent learns that the tiger is an agent-eater. Therefore, the `@BeliefAdded` annotation's pattern represents the action's precondition. In any case, an additional set of annotations for expressing actions directly is also provided. The alternative definition of the `runFromTheTiger` method is shown in Listing 4.6.

With the integration in place, Siebog can be used to deploy intelligent agents with advanced reasoning capabilities, in web and enterprise environments. One practical application of this feature is demonstrated in Section 5.4.

[15] `http://docs.oracle.com/javase/tutorial/reflect/`, retrieved on August 12, 2014.

Listing 4.6: An alternative definition of the agent shown in Listing 4.5, based on action-oriented annotations.

```
public class DNarsExample extends DNarsAgent {
  /* ... */

  @Action(precondition="tiger -> agent_eater (.)")
  public void runFromTheTiger(Statement... reasons) { /* ... */ }
}
```

The remaining agent-oriented programming constructs, such as goals and plans, will be included later once the remaining NAL layers are implemented (Wang, 2012b). More details on further development are given in Chapter 7.

## 4.6 Summary

Siebog is a multiagent middleware that builds on the successes of modern web and enterprise technologies. As shown in this chapter, this design approach has yielded several important features.

On the client, Siebog offers true platform-independence. By running in web browsers, Siebog agents can be executed on a wide variety of hardware and software platforms. This is beneficial to both agent developers, which can write agents in the *write once, run anywhere* manner, and to end-users, which can access their Siebog-based applications in the most convenient way.

The server-side of Siebog runs on top of computer clusters, offering high-availability of deployed applications. The system achieves scalability through automated agent load-balancing, as well as fault-tolerance through state replication and failover.

Moreover, by combining HTML5 and Java EE technologies in a convenient manner, Siebog offers several advanced features, including cross-platform agent interaction, code sharing, and even heterogeneous agent mobility.

Finally, the integration with DNARS enables the deployment of intelligent agents with very useful, practical applications. Case studies that outline the benefits of using the Siebog multiagent middleware are presented in Chapter 5.

Fig. 4.5 outlines the entire technology stack. Contributions of the thesis and our previous research are shown in bold. All underlying technologies: *WildFly*[16], *Apache Cassandra*[17], and *Aurelius Titan*[18] are available as open-source software. DNARS and Siebog (including XJAF and Radigost) are released under the generous Apache License 2.0[19].

[16] `http://wildfly.org/`, retrieved on August 12, 2014.
[17] `http://cassandra.apache.org/`, retrieved on August 12, 2014.
[18] `http://thinkaurelius.github.io/titan/`, retrieved on August 12, 2014.
[19] `http://www.apache.org/licenses/LICENSE-2.0.html`, retrieved on August 12, 2014.

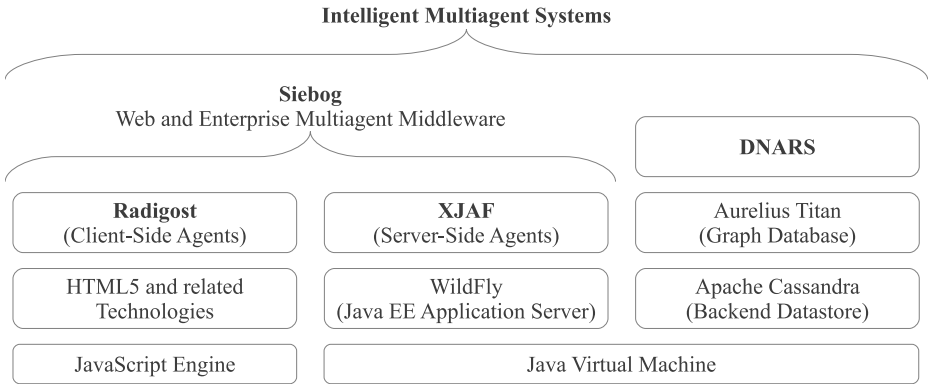



Figure 4.5: The technology stack that forms the basis of DNARS and Siebog, including XJAF and Radigost.
Slika 4.5: Skup tehnologija koji čine osnovu za DNARS i Siebog, uključujući i XJAF i Radigost.

# Chapter 5

# Case Studies

The purpose of this chapter is to present case studies that evaluate and validate the work presented in the previous two chapters. They confirm some of the assertions made earlier and demonstrate practical applications of Siebog. More concretely, three case studies have been developed in order to:

- Demonstrate the Radigost-XJAF integration in practice, in form of a web application that mimics certain functionalities of modern content-sharing networks;
- Evaluate the Resolution engine's responsiveness and scalability, in a scenario which involves a large knowledge base and large numbers of concurrent external clients; and
- Demonstrate how a set of intelligent agents based on DNARS can be used to solve a concrete problem.

## 5.1 Heterogeneous agent mobility and its application

The previous performance evaluation of Radigost presented in (Mitrović et al., 2014a) has shown that the system offers the runtime execution speed comparable to that of a desktop-based multiagent solution. Similarly, it has been shown in (Mitrović et al., 2013b) that XJAF performs better than a third-party multiagent solution for scenarios with large populations of agents. Here, instead of a performance evaluation, one practical application of Siebog will be presented. The case study utilizes the heterogeneous agent mobility which emerges from the Radigost-XJAF integration.

The case study includes a couple of hardware devices; for example, a smartphone and a Smart TV. The user visits the application's web page on the smartphone, takes a photo using the device's camera, and assigns it one or more

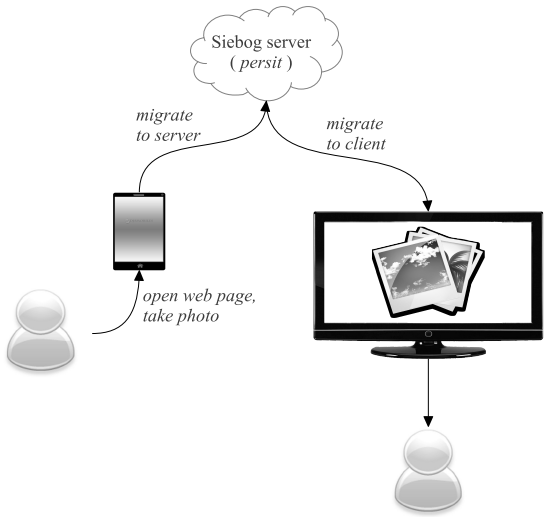



Figure 5.1: Execution flow of the heterogeneous mobility case study.
Slika 5.1: Izvršavanje primera heterogene mobilnosti.

*hashtags*[1]. He/she then activates the mobile agent, which:

- Moves to the server, carrying the photo with it;
- Persists the photo in the database using the set of hashtag(s);
- Moves to the Smart TV, and displays the photo.

The application's execution flow is shown in Fig. 5.1. This case study was partially inspired by the one presented by (Jarvenpaa et al., 2013).

As noted, whenever a web page with Siebog agents is loaded on a client device, the agents and the client device itself are registered with the server. This enables any interested party to inspect and interact with client-side Siebog agents regardless of their physical location. Also, it provides the starting point for agent mobility required here.

The case study consists of the host web page, and the mobile agent. The web page enables the user to take a photo and assign a set of hashtags. It does not require any external plug-ins to take photos, since the media capture and streaming are part of the HTML5 standard[2]. Security is provided at the web browser level: the user is asked whether the application can access the camera.

The *full* source code of the mobile agent, named *PhotoAgent*, is shown in Listing 5.1. Its initialization function `onInit`, receives the photo along with the assigned hashtags. At the end of the initialization phase, the agent moves itself to the server.

[1] `http://en.wikipedia.org/wiki/Hashtag`, retrieved on August 12, 2014.

[2] `http://w3c.github.io/mediacapture-main/getusermedia.html`, retrieved on August 12, 2014.

The migration process is performed as follows:

- Siebog retrieves and remembers the agent's internal state.
- An appropriate REST API call to the WebClient Manager is made, and the state is transferred to the server.
- On the server, the state is injected into an instance of the *RadigostAgent* component. RadigostAgent uses the Java Scripting API to execute and interact with the embedded JavaScript code.
- The system invokes the agent's `onArrived` function and the agent continues its execution.

Listing 5.1: The full source code of the *PhotoAgent* used in the heterogeneous mobility case study. This mobile agent moves between client devices and the server, carrying the user's photo with it.

```
importScripts("/siebog/radigost.js");
function PhotoAgent() { };
PhotoAgent.prototype = new Agent();

PhotoAgent.prototype.onInit = function(args) {
  this.photo = args.photo;
  this.hashtags = args.hashtags;
  this.moveToServer();
};

PhotoAgent.prototype.onArrived = function(hap, isServer) {
  if (isServer) {
    importClass(Packages.siebog.agents.radigost.photo.PhotoAgentJPA);
    var jpa = Packages.siebog.agents.radigost.photo.PhotoAgentJPA;
    var destClients = jpa.persist(this.hashtags, this.photo);
    this.moveToClients(destClients);
  } else // on the dest client, show the photo
    this.onStep(this.photo);
};
```

On the server, the agent uses a helper component named *PhotoAgentJPA*. The component is a wrapper around the *Java Persistence API*[3], providing the higher-level access to the backend database. The agent uses it to store the photo and to retrieve all destination clients that are observing the hashtags. Finally, the agent moves (in parallel) to each destination client and shows the photo.

In conclusion, this case study demonstrates the benefits of combining agents with HTML5 and Java EE technologies. It offers functionalities similar to many modern web applications, such as social and content sharing networks, but conveniently uses mobile agents to deliver information across remote client devices.

[3] `http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`, retrieved on August 12, 2014.

Therefore, the Siebog multiagent middleware can seamlessly integrate software agents into modern web and enterprise applications.

In the remaining two case studies, the focus is on intelligent DNARS agents within the Siebog framework. The first case study of the two will evaluate the Resolution engine's runtime efficiency, and requires a large backend knowledge base.

## 5.2 Creating a large knowledge base for DNARS

The large knowledge base needed for the second case study has been extracted from the *DBpedia* datasets, available in form of *Resource Description Framework* (RDF) statements. This section provides more details about the RDF data model, as well as the motivation behind the DBpedia project and its end-goals.

### 5.2.1 Resource Description Framework

RDF is one of the most popular knowledge representation and sharing standards, widely used in e.g. the *Semantic Web*[4]. It represents a data model for describing and inter-linking (most-commonly) web resources (Brickley and Guha; Schreiber and Raimond).

RDF includes a human- and machine-readable language with a formal grammar and an accompanying set of tools for writing, querying, analyzing, etc. RDF-based data. The RDF-based data can be written using several notations and can be serialized in a number of ways. RDF-related specifications are published and maintained by the *World Wide Web Consortium*[5].

The three main data types that can be used in RDF are *International Resource Identifiers* (IRIs), *literals*, and *blank nodes* (Schreiber and Raimond). IRIs are globally-unique identifiers. They are based on the more-common *Uniform Resource Identifiers* (URIs)[6], but can also include non-ASCII characters. An example of an IRI denoting the famous scientist Albert Einstein is:[7]
`http://dbpedia.org/resource/Albert_Einstein`.

RDF literals are primitive types, and are used to represent numbers, strings, etc. For a more convenient interpretation, the literal can be associated with the concrete data type identifier. Furthermore, string literals can be marked using a language identifier. Finally, blank nodes are used in RDF to denote *anonymous* resources, i.e. resources that are not explicitly represented by an IRI.

RDF uses *statements* to describe *resources*. A statement is a triplet in the form of *subject-predicate-object*. It describes a relationship between the two resources (Schreiber and Raimond). In a statement the subject can be an IRI or a blank node, the predicate must be an IRI, while the object can be an IRI,

[4] `http://www.w3.org/standards/semanticweb/`, retrieved on August 12, 2014.
[5] `http://www.w3.org/RDF/`, retrieved on August 12, 2014.
[6] `http://tools.ietf.org/html/rfc3986`, retrieved on August 12, 2014.
[7] The example is taken from a *DBpedia* dataset, described in Sub-section 5.2.2.

a literal, or a blank node. For example, in the following statement, subject and predicate are IRIs, while the object is a string literal:

```
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/birthDate>
    "1879-03-14"^^http://www.w3.org/2001/XMLSchema#date .
```

The given statements describes Albert Einstein's date of birth. The object is thus a string literal associated with the *date* type.

RDF uses a *Schema* to structure and provide semantic information about resources (Brickley and Guha). RDF Schema represents the way of building the so-called *vocabularies* (or *ontologies*) to be used alongside the actual data. Its terminology is based on the one used in object-oriented programming and includes *classes*, *inheritance*, *properties*, etc. For example, the following statement can be used to describe that *Albert Einstein* is an instance (or a *type*) of the class *Person*:

```
<http://dbpedia.org/resource/Albert_Einstein>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
```

As noted, RDF-based data can be written using many different notations. The examples given above use the *N-Triples* notation, which is a simple, text-based format that allows for fast and easy parsing. Descriptions of other popular formats can be found in e.g. (Schreiber and Raimond).

RDF also offers the way for querying the data. *SPARQL* is a standard[8] language for querying and updating RDF-based datasets. Its syntax is loosely-based on SQL, with the support for joins, sorting, aggregation, etc. Listing 5.2 shows a simple SPARQL query that retrieves the Einstein's birth place.

Listing 5.2: A SPARQL query that retrieves information about Albert Einstein's birth place.

```
PREFIX resource: <http://dbpedia.org/resource/>
PREFIX ontology: <http://dbpedia.org/ontology/>
SELECT ?birthPlace
WHERE { resource:Albert_Einstein ontology:birthPlace ?birthPlace }
```

As it can be seen, NAL and RDF use the same form of subject-predicate-object statements. This is why RDF has been chosen as the main data model for the remaining two case studies.

A large number of publicly-available RDF-based datasets can be found online. Datasets used in this section have been retrieved from the *DBpedia* project.

[8] `http://www.w3.org/2011/05/sparql-charter`, retrieved on August 12, 2014.

### 5.2.2 DBpedia

*DBpedia* (Lehmann et al., 2014) is a community-driven project aimed at organizing and structuring the information extracted from the free *Wikipedia* encyclopedia[9]. The project's goal is to "... make it easier for the huge amount of information in Wikipedia to be used in some new interesting ways. Furthermore, it might inspire new mechanisms for navigating, linking, and improving the encyclopedia itself." [10]

At the time of writing, the latest version of DBpedia is 3.9, published in 2014[11]. Its English version describes 4.58 million entities, such as persons, organizations, places, species, etc. The data is also provided in 125 different languages, which, together with the English version, sums up to 38.3 descriptions. A large portion of these information is classified using a manually-created, consistent ontology, which defines 685 classes and 2795 properties. All the data is licensed under the terms of the *Creative Commons Attribution-ShareAlike 3.0*[12] and the *GNU Free Documentation*[13] licenses.

DBpedia information is available in form of RDF statements. Each DBpedia entity is identified using an IRI, which is derived from the corresponding Wikipedia entry. The data is organized into a number of datasets[14]. For the remaining DNARS-related case studies, three datasets were of the special interest. The *Short Abstracts* dataset contains approximately 4.5 million short abstracts of Wikipedia articles. The *Mapping-based Types* dataset describes types/classes of approximately 28 million entities.

Finally, *Mapping-based Properties (Cleaned)* includes information extracted from Wikipedia *infoboxes*. Here, the infobox is a summary of the entire article, and contains important facts and statistics. It is usually present in form of a table on the right side of the article. The dataset is cleaned and improved through the use of heuristic inference (Paulheim and Bizer, 2014). The 2014 version includes approximately 26 million RDF statements.

In essence, each DBpedia dataset contains different information (or different form of information) about a particular entity. As an example, Listing 5.3 shows how Albert Einstein is described in each of the three datasets.

DBpedia is being actively developed and improved, and has inspired several interesting projects. For example, the research presented by (Mendes et al., 2012) shows how DBpedia datasets can be used to improve natural language processing. Similarly, the *DBpedia Spotlight* project (Mendes et al., 2011) can be used to identify DBpedia resources in unstructured texts.

---

[9] `http://www.wikipedia.org/`, retrieved on August 12, 2014.

[10] `http://dbpedia.org/About`, retrieved on August 12, 2014.

[11] `http://blog.dbpedia.org/2014/09/09/dbpedia-version-2014-released/`, retrieved on August 12, 2014.

[12] `http://creativecommons.org/licenses/by-sa/3.0/legalcode`, retrieved on August 12, 2014.

[13] `http://www.gnu.org/copyleft/fdl.html`, retrieved on August 12, 2014.

[14] `http://wiki.dbpedia.org/Datasets`, retrieved on August 12, 2014.

Listing 5.3: Descriptions of Albert Einstein in DBpedia datasets.

```
/*** Mapping-based Properties (Cleaned) ***/
<http://dbpedia.org/resource/Albert_Einstein>
  <http://xmlns.com/foaf/0.1/name>
  "Albert Einstein"@en .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/birthDate>
  "1879-03-14"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/birthPlace>
  <http://dbpedia.org/resource/German_Empire> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/residence>
  <http://dbpedia.org/resource/Switzerland> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/residence>
  <http://dbpedia.org/resource/United_States> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/spouse>
  <http://dbpedia.org/resource/Mileva_Mari%C4%87> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/field>
  <http://dbpedia.org/resource/Physics> .
/*** Short Abstracts ***/
<http://dbpedia.org/resource/Albert_Einstein>
    <http://www.w3.org/2000/01/rdf-schema#comment> "Albert Einstein (/\u
    02C8\u 00E6lb\u 0259rt \u 02C8a\u 026Ansta\u 026An/; German: [\u
    02C8alb\u 0250t \u 02C8a\u 026An\u 0283ta\u 026An] ; 14 March 1879
    \u 2013 18 April 1955) was a German-born theoretical physicist. He
    developed the general theory of relativity, one of the two pillars
    of modern physics (alongside quantum mechanics). He is best known
    for his mass\u 2013energy equivalence formula E = mc2 (which has
    been dubbed \'the world's most famous equation\')."@en .
/*** Mapping-based Types ***/
<http://dbpedia.org/resource/Albert_Einstein>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/ontology/Scientist> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://wikidata.dbpedia.org/resource/Q5> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://www.w3.org/2002/07/owl#Thing> .
<http://dbpedia.org/resource/Albert_Einstein>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://schema.org/Person> .
```

## 5.3 Evaluating the speed of question answering

As discussed in Chapter 3, the Resolution engine of DNARS is responsible for answering questions. For questions containing "?" (i.e. *S copula* ? or ? *copula P*), the engine returns the best possible candidate for the missing term. For questions in the form of "*S copula P*" it checks whether the corresponding statement exists in the knowledge base, or whether it can be derived using NAL's backward inference rules.

The backward inference engine can take an undetermined amount of time to execute (Wang, 2013). On the other hand, one of the functional requirements of DNARS is to answer the first type of questions as quickly as possible, in real-time. The following case study has been designed to evaluate this capability of the Resolution engine.

The DBpedia dataset used in this case study is *Mapping-based Properties (Cleaned)* described earlier. Its RDF statements have been imported into the DNARS Backend knowledge base by using arbitrary relations of NAL-4 discussed in Section 2.6. For example, the following RDF statement describes one property of Albert Einstein:

```
<http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/field>
  <http://dbpedia.org/resource/Physics> .
```

In the given statement, the first line represents the subject (*Albert Einstein*), the second line represents the predicate (his *field* of study), while the third line represents the object (*physics*). The field of study is the arbitrary relation, so the corresponding NAL-4 statement is written as follows:

```
(× <http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/ontology/field>) →
  <http://dbpedia.org/resource/Physics>
```

As noted in Section 2.6, in order to more easily apply inference rules, the NAL-4 statement can be structurally transformed into extensional and/or intensional images. These images are just different forms of the same statement; they show how individual atomic terms from the original statement link to the remaining parts of the statement. For the statement given above, the two extensional images are written as follows:

```
<http://dbpedia.org/resource/Albert_Einstein> →
  (/ <http://dbpedia.org/ontology/field> ◇
  <http://dbpedia.org/resource/Physics>)
<http://dbpedia.org/resource/Physics> →
  (/ http://dbpedia.org/ontology/field>
  <http://dbpedia.org/resource/Albert_Einstein> ◇)
```

Instead of creating images at runtime, DNARS stores three NAL statements per one RDF statement. This design approach was made in order to improve

the runtime efficiency of the Resolution engine. It represents a standard practice – systems that work with NoSQL databases often repeat the stored information in order to improve their runtime efficiency (Schram and Anderson, 2012).

As noted in Subsection 3.2.2, NAL-based knowledge bases are actually property graphs: directed multi-relational graphs with any number of properties attached to vertices and edges (Robinson et al., 2013; Rodriguez and Shinavier, 2010). Once the entire *Mapping-based Properties (Cleaned)* dataset was imported into DNARS, the resulting graph consisted of approximately 60 million vertices and 77 million edges. According to today's standards, the graph can be called *large* (e.g. McColl et al., 2014).

### 5.3.1 Speed benchmarks

The experiments were performed in clusters provided by the *Microsoft Azure* cloud computing platform[15]. Two types of machines were used (both using an SSD storage):

- *D3*: 4 virtual CPUs, 14 GB of RAM.
- *D4*: 8 virtual CPUs, 28 GB of RAM.

In order to simulate large numbers of concurrent users, the *Yahoo! Cloud Serving Benchmark* (YCSB) was used (Cooper et al., 2010; Kuhlenkamp et al., 2014). YCSB is an open-source tool[16] designed for load-testing of (primarily) NoSQL databases. It can be configured through a range of parameters, most important of which is the desired number of operations per second (throughput), but also the number of concurrent threads, maximum execution time, etc.

Two types of scenarios were examined: read-only and read-write. In the read-only scenario, clients only ask questions and no writing to the DNARS knowledge base is performed. The read-write scenario, on the other hand, is more realistic (and computationally more demanding), since some clients ask questions, while others add new knowledge to the system.

Within each scenario, DNARS was deployed on three different hardware configurations. The goal was to determine how the underlying hardware affects the system's performance.

YCSB client was executed on a separate D4 machine, and was configured to use 100 threads. The CPU utilization on the client machine was never over 20%, so it did not represent the bottleneck.

The YCSB client executed a number of test-cases, each lasting for 1 hour. The desired throughput (i.e. the number of questions per second) was increased for each test-case, until the system could not reach it anymore. The efficiency of DNARS is expressed in terms of average, 95th percentile and 99th percentile latencies. The later two values indicate the maximum latencies exhibited by, respectively, 95% and 99% of clients (Cooper et al., 2010; Kuhlenkamp et al., 2014).

[15] `http://azure.microsoft.com/en-us/`, retrieved on August 12, 2014.
[16] `https://github.com/brianfrankcooper/YCSB/`, retrieved on August 12, 2014.

Finally, DNARS was restarted before each test-case. Questions were constructed by selecting random statements from the dataset. Approximately 80% of questions that were asked were new, while the remaining 20% were repeated questions. This put an additional strain on the system, as it could not fully benefit from answer caching.

The question answering capabilities of DNARS in the read-only scenario are shown in Fig. 5.2. More specifically, Fig. 5.2(a) shows the performance of DNARS on a single D3 node, Fig. 5.2(b) shows its performance on a single D4 node, while Fig. 5.2(c) shows how the system performs when it's distributed over two D3 nodes.

The obvious conclusion for all three configurations is that DNARS performs exceptionally well. On the lowest hardware configuration (Fig. 5.2(a)), the system is capable of answering almost 5800 questions per second, with the 99th percentile latency being 100 milliseconds. Once the number of virtual CPUs is doubled (Fig. 5.2(b)), the maximum number of answers per second jumps to over 9200, with 99% clients having to wait no more than 30 milliseconds. In the final hardware configuration (Fig. 5.2(c), two D3 machines), DNARS can provide answers to approximately 8300 questions per second, in which case the 99th percentile latency is just over 50 milliseconds.

The underlying Apache Cassandra database was obviously an excellent choice for the backend storage, as it can efficiently use all the available hardware resources. Vertical scaling (i.e. adding more virtual CPUs) yields better performance than horizontal scaling (i.e. adding more machines). However, in addition to practical limitations of vertical scaling, horizontal scaling has one major advantage – it can provide fault-tolerance through data replication.

For the second, read-write scenario, an additional YCSB client was launched on a separate machine. Its task was to add 100 statements per second to the DNARS knowledge base, throughout the duration of the experiment. Moreover, it added only statements that already existed in the knowledge base. This is because adding an existing statement is slower than adding a new statement. In the first case, the system needs to read the existing truth-value from the harddisk, perform revision, and write the new value back (which will also update the database indexes).

Again, three different hardware configurations were deployed – one D3, one D4, and two D3 machines – and the results are shown in Fig. 5.3. Obviously, simultaneous writing to the database incurs some runtime penalty, and the latencies are generally higher than in the read-only scenario. Nonetheless, the results can still be considered excellent.

The most affected configuration is the single D3 node (Fig. 5.3(a)), but it can still deliver 4400 answers per second, with 99% clients having to wait up to 100 milliseconds. The simultaneous writing to the database did not affect the D4 node as much, since it was still capable of answering over 8800 questions per second, while keeping the 99th percentile latency at 30 milliseconds. Finally, when distributed over two D3 nodes, in the read-write scenario DNARS can answer 7700 questions per second with the 99th percentile latency at 60 milliseconds.

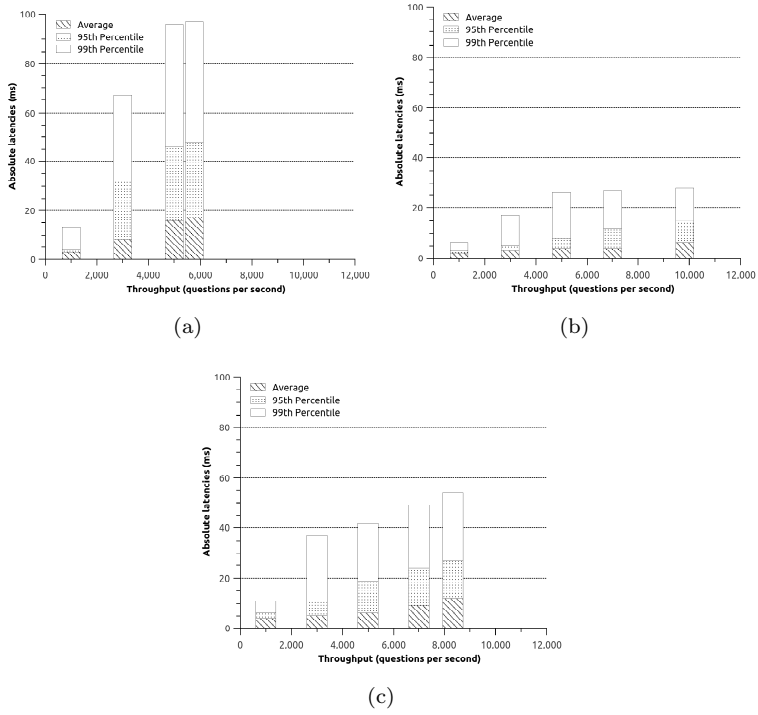
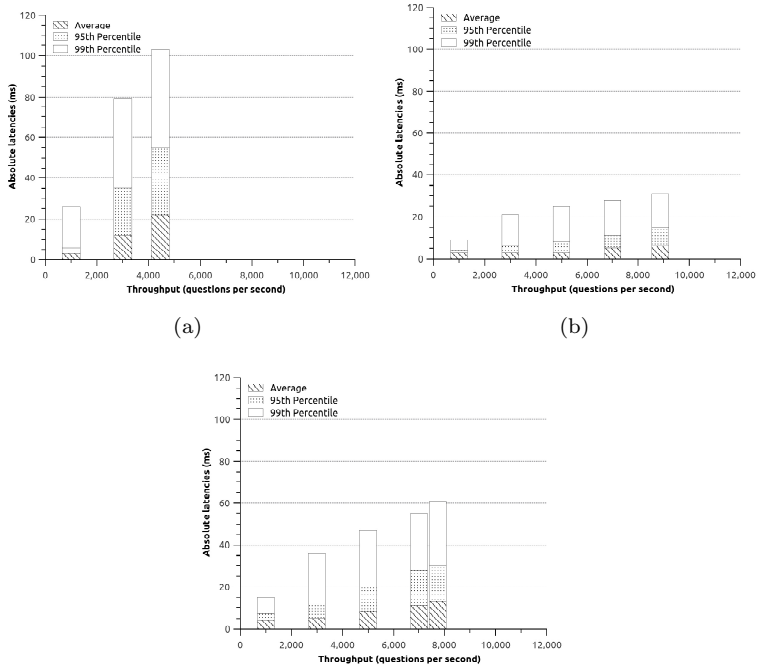



Figure 5.2: Runtime performance of DNARS in the read-only scenario, on (a) a single D3 node, (b) a single D4 node, and (c) two D3 nodes.
Slika 5.2: Performanse izvršavanja DNARS-a u scenariju sa samo čitanjem podataka, na (a) jednom D3 čvoru, (b) jednom D4 čvoru, i (c) dva D3 čvora.




(a) (b)

(c)

Figure 5.3: Runtime performance of DNARS in the read-write scenario, on (a) a single D3 node, (b) a single D4 node, and (c) two D3 nodes.
Slika 5.3: Performanse izvršavanja DNARS-a u scenariju sa pisanjem i čitanjem podataka, na (a) jednom D3 čvoru, (b) jednom D4 čvoru, i (c) dva D3 čvora.

These experiments confirm that the functional requirement imposed on the Resolution engine in Chapter 3 has been fulfilled. That is, the engine is capable of supporting a large knowledge base and providing real-time responses to high numbers of external clients.

## 5.4 Deriving new knowledge for DBpedia

One final question still remains – can intelligent agents based on the current implementation of DNARS solve a concrete practical problem? The third case study provides an affirmative answer and validates the overall work of the thesis.

The main goal of this case study is to derive new structured knowledge base for DBpedia using information available in unstructured texts. It relies on the three DBpedia datasets described earlier. More concretely, the case study derives new knowledge for the *Mapping-based Properties (Cleaned)*, using information in *Short Abstracts* and *Mapping-based Types*.

The case study is shown graphically in Fig. 5.4. Its execution sequence can be described in 5 distinctive steps.

**Step 1.** The case study is started by an end-user, who asks a question about a specific resource. The question ends up in the *Resolver* agent, which returns all the information available in *Short Abstracts* and *Mapping-based Properties (Cleaned)*. The agent relies on the Resolution engine to find the required answers.

**Step 2.** Once the answers are returned the Resolver activates the *Annotator* agent. This new agent annotates the unstructured text obtained from *Short Abstracts*, by invoking the DBpedia Spotlight RESTful web service[17]. In response, the agent receives a list of DBpedia resources found in the text. Now, the system needs to determine the exact relations between the properties of the initial resource and the received annotated resources.

**Step 3.** For each annotated resource, the Annotator creates an instance of the *Learner* agent. The Learner agent first retrieves all statements relevant to its resource. Relevant statements are answers to questions $R \rightarrow ?$ and $? \rightarrow R$, where $R$ denotes the agent's resource. The answers are retrieved from the *Mapping-based Properties (Cleaned)* dataset.

**Step 4.** Now, the Learner agent employs the Forward inference engine to derive *intermediary* conclusions. Known properties of the initial resource are used as the knowledge base, while the relevant statements represent new judgments. Intermediary conclusions derived in this step serve as initial links between properties of the initial resource and properties of annotated resources.

[17] `http://spotlight.dbpedia.org/rest/annotate`, retrieved on August 12, 2014.

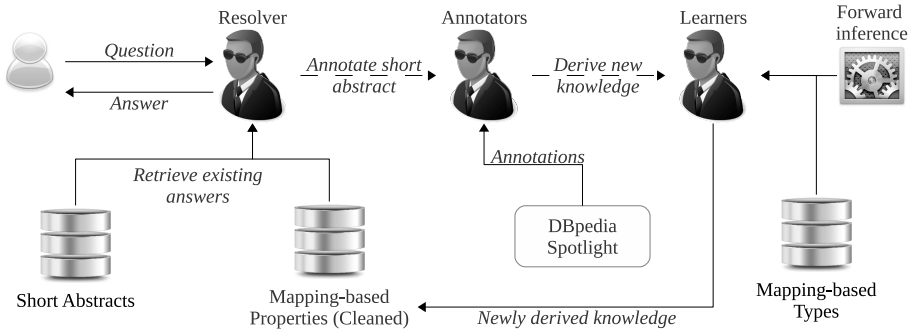



Figure 5.4: Execution flow of the case study for deriving new structured knowledge.
Slika 5.4: Izvršavanje primera izvođenja novog struktuiranog znanja.

**Step 5.** Finally, intermediary conclusions obtained in the previous step are again matched against properties of the initial resource, to derive the set of conclusions. This set is first filtered, merging duplicate statements using the revision rule (Section 2.8), and resulting in the final, new structured knowledge about the initial resource.

In this multiagent system, the Resolver and Learners represent true intelligent, deliberative agents, with respect to the definition outlined at the end of Section 1.1. That is, both agents use a symbolic model of the world and apply logical reasoning to answer questions and derive new knowledge. The Annotator is a simpler, reactive XJAF agent with the task of invoking a web service and distributing the computational load across the computer cluster.

### 5.4.1 A concrete execution example

In this sub-section, we will illustrate the above steps on a concrete example. Let the end-user ask: *Albert_Einstein* $\rightarrow$ ? ("Who was Albert Einstein?"). In Step 1, the Resolver uses the Resolution engine to retrieve the short abstract and the list of existing properties from the knowledge base. In Step 2, the Annotator sends the short abstract to the DBpedia Spotlight web service and receives the four DBpedia resources shown in Listing 5.4[18].

Listing 5.4: Annotated resources detected in the short abstract for *http://dbpedia.org/resource/Albert_Einstein*.

```
<http://dbpedia.org/resource/General_relativity>
<http://dbpedia.org/resource/Max_Born>
<http://dbpedia.org/resource/Quantum_mechanics>
<http://dbpedia.org/resource/Theoretical_physics>
```

[18]Only resources available in the *Mapping-based Properties (Cleaned)* dataset are shown.

At this point the system knows that these resources are somehow related to Albert Einstein. Now it needs to determine the exact relations. This is initiated in Step 3. A new Learner agent is created for each annotated resource in Listing 5.4. The agent first retrieves statements relevant to its resource. Then, in Step 4, it uses forward inference with known properties of Albert Einstein as the knowledge base, and the relevant statements as new input judgments. This step derives a set of intermediary conclusions (the total of 249 for all Learners). For example, intermediary conclusions include 13 statements stating that general relativity is similar to physics, that is:

<http://dbpedia.org/resource/General_relativity> $\leftrightarrow$
  <http://dbpedia.org/resource/Physics> $\langle 1.00, 0.45 \rangle$

In the final step (Step 5) each Learner again applies the forward inference using known properties of Albert Einstein as the knowledge base and intermediary conclusions now as new input judgments. Conclusions derived in this step are first filtered to merge duplicate statements and to exclude already known properties.

The end-result – the newly derived structured knowledge about Albert Einstein – is shown in Listing 5.5. Only statements with the confidence level of 0.9 or higher are taken into account as this is the value assigned to existing statements (Wang, 2013).

Listing 5.5: The newly derived structured knowledge about Albert Einstein.

```
(× <http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/resource/General_relativity>) →
    <http://dbpedia.org/ontology/field> ⟨1.00, 0.90⟩

(× <http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/resource/Quantum_mechanics>) →
    <http://dbpedia.org/ontology/field> ⟨1.00, 0.92⟩

(× <http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/resource/Theoretical_physics>) →
    <http://dbpedia.org/ontology/field> ⟨1.00, 0.99⟩
```

Manual inspection of these results confirms that they are correct. General relativity, quantum mechanics, and theoretical physics were indeed Einstein's fields[19]. This information is present in the *Short Abstracts* but not in the *Mapping-based Properties (Cleaned)* dataset (Listing 5.3) and represents new structured knowledge.

The fourth annotated resource denoting the physicist Max Born could not be linked to Einstein in a confident manner. The *Mapping-based Properties (Cleaned)* dataset version 3.9 (2014) includes the total of 14 statements about Albert Einstein. These statements use the following set of relations: *doctoral*

[19]However, he was displeased with the principles of quantum mechanics and was trying to disprove the theory (Kumar, 2009).

*advisor*, *academic advisor*, *name*, *birth place*, *birth date*, *death place*, *death date*, *residence*, *spouse*, and *field*. Since at the current level DNARS cannot derive new relations, it has incorrectly concluded that Born was Einstein's doctoral advisor[20]. Although incorrect, it would have been much worse if the system had derived:

```
(× <http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/resource/Max_Born>) →
    <http://dbpedia.org/ontology/birthPlace> ⟨1.00, 0.90⟩
```

Therefore, DNARS has still selected the arguably best possible relation among the available ones. It is also worth noting that this conclusion had a lower confidence value then the required 0.9.

### 5.4.2 Analysis of the reasoning process

Let us now analyze how the first conclusion in Listing 5.5 was derived. At some point during the reasoning process, the Forward inference engine takes the two premises shown in Listing 5.6.

Listing 5.6: The initial premises that will lead to the conclusion that General relativity was Einstein's research field.

```
(× <http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/resource/Physics>) →
    <http://dbpedia.org/ontology/field>

(× <http://dbpedia.org/resource/Charles_W._Misner>
  <http://dbpedia.org/resource/General_relativity>) →
    <http://dbpedia.org/ontology/field>
```

By intensional comparison (Eq. 2.14), the system derives a similarity between two compound terms shown as the first statement in Listing 5.7. Since the relation between compound terms defines relations between their respective components, this statement can be transformed into the latter two intermediary conclusions in Listing 5.7. The *Mapping-based Types* dataset is employed during this transformation step. The transformation will be applied only if the related components belong the same type, preventing the system to conclude that, for example, a person is similar to a geographic location.

This process of deriving and transforming intermediary conclusions is repeated for many other physicists in the *Mapping-based Properties (Cleaned)* dataset, and the revision rule steadily increases the system's confidence about the fact that General relativity is similar to Physics. On the other hand, the similarity between Charles W. Misner and Albert Einstein is derived only once, retaining a relatively low confidence (although the two are similar to some respect).

[20]The two famous physicist were, however, colleagues and friends (Born, 1971).

Listing 5.7: The first statement represents an intermediary conclusion derived by applying intensional comparison to the premises from Listing 5.6, while the latter two statements are obtained by transforming the first one.

```
(× <http://dbpedia.org/resource/Charles_W._Misner>
  <http://dbpedia.org/resource/General_relativity>) ↔
    (× <http://dbpedia.org/resource/Albert_Einstein>
      <http://dbpedia.org/resource/Physics>)

<http://dbpedia.org/resource/Charles_W._Misner> ↔
  <http://dbpedia.org/resource/Albert_Einstein>

<http://dbpedia.org/resource/General_relativity> ↔
  <http://dbpedia.org/resource/Physics>
```

In the final inference step the system uses the analogy rule (Eq. 2.17) on the known statement:

```
(× <http://dbpedia.org/resource/Albert_Einstein>
  <http://dbpedia.org/resource/Physics>) →
    <http://dbpedia.org/ontology/field>
```

or its extensional image:

```
<http://dbpedia.org/resource/Physics> →
  (/ <http://dbpedia.org/ontology/field>
    <http://dbpedia.org/resource/Albert_Einstein> ◇)
```

and the intermediary conclusion:

```
<http://dbpedia.org/resource/General_relativity> ↔
  <http://dbpedia.org/resource/Physics>
```

to derive the final, highly-confident conclusion that General relativity was an additional research field of Albert Einstein.

## 5.5 Summary

In order to validate the work presented in this thesis, three case studies have been presented.

The first case study demonstrates how the Siebog multiagent middleware can be used to integrate agents into modern web applications. It exploits the support for heterogeneous agent mobility in order to provide functionalities commonly found in modern social and content-sharing web applications. Therefore, it conveniently bridges the gap between the agent technology and the industry.

The second case study was developed in order to evaluate the responsiveness of the Resolution engine. Its results have shown that the engine can support large numbers of external clients, even with a large backend knowledge base. It

performed exceptionally well in all three hardware configurations, both in the read-only and read-write scenarios.

Finally, the third case study has shown how intelligent agents based on DNARS can be used to solve a concrete problem. A set of both intelligent and reactive agents was deployed in a distributed setting in order to generate new structured knowledge. In the end, the case study has shown how an AGI system, built by combining the agent technology with DNARS, can be used to make contributions to the community-driven DBpedia project.

In the next, and final part, the focus will be on analyzing the existing, related work. The overall conclusions and planned future research directions will be discussed as well.

# Part III

# Related and Future Work

# Chapter 6

# Related Work

Non-axiomatic logic (NAL) is different from many other logics used by the agent technology or artificial (general) intelligence researchers. Its differences stem from the fact that NAL is a term logic with syllogistic inference rules and the experience-grounded semantics. In-depth comparisons of NAL and other formalisms can be found in many existing research papers (e.g. Wang, 2000, 2001a, 2006, 2011, 2012a, 2013).

This chapter provides a comparison of the work presented in this thesis with other relevant work. Section 6.1 discusses the advantages (and disadvantages) of Siebog when compared to other existing multiagent middlewares without concentrating on agents' reasoning capabilities. Section 6.2 focuses on existing reasoning and cognitive architectures, both general-purpose and BDI-based, and compares them with DNARS.

## 6.1 Multiagent middlewares

Most existing multiagent middlewares have been designed as desktop or server-side systems, with the addition of various *bridges* (in form of Java Applets, for example), that enable web access (Jack WebBot; Kelemen, 2006; Minotti et al., 2010). The first part of this section analyzes existing web-enabled multiagent middlewares, while in the second part the focus is solely on server-side functionalities.

### 6.1.1 Web-based multiagent middlewares

There exist several interesting multiagent middlewares that enable web-based clients. *JACK Intelligent Agents WebBot* (Jack WebBot), for example, is based on *Java Servlet*[1] and *JavaServer Pages*[2] technologies. It provides a link between

[1] `http://www.oracle.com/technetwork/java/index-jsp-135475.html`, retrieved on August 12, 2014.

[2] `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`, retrieved on August 12, 2014.

web clients and JACK agents running in a web server.

Similarly, *JadeGateway* and *GatewayAgent* classes (Kelemen, 2006) enable web clients to communicate with JADE agents hosted on a remote server. The main disadvantage of these approaches, when compared to Siebog, is that their agents are executed on a web server, rather than on clients devices. Large numbers of hosted agents and client requests can pose significant computational loads on the server. Instead, the usage of Web Workers for delegating tasks to remote clients can substantially reduce the server's computational load (Okamoto and Kohana, 2010).

*Smart Python multi-Agent Development Environment* (SPADE) is a multi-agent platform characterized by the usage of *XMPP/Jabber* instant messaging protocol[3] for agent communication (Aranda et al., 2012). XMPP/Jabber enables the system to employ "an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML" (Aranda et al., 2012). SPADE supports the client-server architecture, with a designated server machine, and agents distributed across remote client devices.

*JaCa-Web* is a multiagent platform aimed at developing agent-based web applications (Minotti et al., 2010). It is built on top of JaCa, which consists of the Jason interpreter (Bordini and Hubner, 2006; Bordini et al., 2007), and the CArtAgO framework for artifact modeling (Boissier et al., 2013). JaCa-Web agents and artifacts are hosted and executed inside the client's web browser. The platform includes two predefined artifacts: *Page* for two-way communication between agents and the web page, and *HTTPService* for remote HTTP service invocation.

JaCa-Web brings the full power of Jason and CArtAgO to web applications, and can currently offer a much wider range of functionalities than Siebog. The main advantage Siebog has over JaCa-Web is its platform independence described earlier. More concretely, JaCa-Web requires Java runtime environment to be present on the client device. Examples found on the framework's web site[4] were not able to run on an *Apple iOS*-powered smartphone device, which was perfectly capable of running Siebog agents (Mitrović et al., 2014a).

The main difference between Siebog and all of these middlewares is that its client side (i.e. Radigost) is developed in the manner of modern web applications: using the HTML5 set of technologies. An important advantage of this approach is greater platform-independence: Siebog is the only system among this group that requires no virtual machine or browser plug-in to run. For example, in order to run a SPADE agent, the client must be equipped with the Python runtime environment, which might not be available for all platforms. Siebog is also readily available to end-users, without any installation or configuration steps. Finally, unlike JACK WebBot and JadeGateway, Siebog agents are actually executed on the client side, reducing the server load.

At the time of writing, the only other purely HTML5-based agent platform

[3] `http://www.jabber.org/`, retrieved on August 12, 2014.

[4] `http://jaca-web.sourceforge.net/`, retrieved on August 12, 2014.

is described by (Jarvenpaa et al., 2013). There are several important differences between the two systems. Siebog is more advanced on the client side, as it fully utilizes the advantages of Web Workers and the WebSocket protocol. On the server side, while Siebog relies on Java EE, their platform conveniently uses *Node.js*, a JavaScript-based server framework[5]. Although this approach simplifies the implementation of certain functionalities, such as agent mobility, it lacks the cluster-based features of Java EE available in Siebog.

### 6.1.2 Comparing the server-side features of Siebog

Agentis is one of the earliest multiagent architectures (D'Inverno et al., 1998; Kinny, 1999), built on top of the BDI reasoning engine dMARS (D'Inverno et al., 2004). It placed a great emphasis on agent interaction protocols, which were designed as reliable and efficient, supporting multiple concurrent executions. Agents were organized in a hierarchical fashion and communicated using a strongly-typed language. Their capabilities were expressed in terms of services, complex activities initiated by external clients and tasks, simpler activities used to realize services. Unfortunately, the system does not appear to be developed or maintained anymore.

Currently, there are several multiagent middlewares that offer agent load-balancing and/or fault-tolerance. *Cognitive Agent Architecture* (Cougaar) is a Java-based distributed agent architecture specifically designed for unstable environments (BBN, 2004; Siracuse et al., 2007). Cougaar provides state persistence and error recovery for its agents. Since its internal components are designed as agents they too are protected by the fault-tolerance sub-system. Agent distribution and fault-tolerant features in Cougaar are more powerful than those found in Siebog. However, the development of Siebog demonstrates how many of these features can be realized with much less effort and much fewer resources, by using standard and ready-made solutions in Java EE.

*Magentix* is a Linux-based multiagent middleware. It is built with the runtime performance as the primary focus (Alberola et al., 2013). For this purpose, the system is heavily based on low-level features offered by the operating system. For example, each agent is represented by a Linux process with three internal threads. The platform itself can be distributed across a number of computers. Although it achieves remarkable runtime performance, Magentix lacks previously described features of XJAF that stem from the use of computer clusters.

JADE is a popular, Java-based multiagent middleware (Bellifemine et al., 2005, 2007). It supports the development of both reactive and cognitive agents, and features an extensive ecosystem of plug-ins. JADE's agent containers can be distributed across a computer network, and has a support for fault-tolerance at the container level.

There are many differences between inner workings of JADE and Siebog. By analyzing the source code of Siebog one can conclude that there is no messaging

[5] `http://nodejs.org/`, retrieved on August 12, 2014.

infrastructure implementation; instead, the underlying *Java Message Service*[6] is used. The biggest difference is that JADE agents have to be manually distributed among the containers, whereas in Siebog this process is performed automatically. More concretely, a Siebog agent lives on top of the entire cluster and not on a single computer. When it needs to process a message it can do so on any computer available. As shown by (Mitrović et al., 2014b), Siebog represents a better solution for applications that need to launch large populations of agents (e.g. Ilie et al., 2011), and/or need to provide high-level of fault-tolerance. For example, JADE consumes a single thread per agent and has a pre-fixed number of message processing threads. In Siebog, these numbers are increased or decreased automatically depending on the current load.

So far only a few agent middlewares have been built using Java EE. *Agent Developing Framework* (Nichifor and Buraga, 2004) employed a minimum set of Java EE technologies for some of its functionalities but does not seem to be developed anymore. *Voyager*[7] is a commercial product and more of an enterprise middleware with agent support, than a fully-featured multiagent framework.

*Whitestein LS/TS* represents a comprehensive set of development tools, a UML-based modeling language and a high-level Java library for writing and deploying agents (Rimassa et al., 2005). It is offered in three different editions – *Personal*, *Business*, and *Enterprise*, with the first two running on Java SE and the third one employing Java EE technologies. Due to the high-level library an agent is written only once and can run on any of the editions. The Enterprise edition can be run on top of a computer cluster in order to provide fault-tolerance. However, since this edition is a commercial product, a deeper comparison with Siebog could not be provided.

Finally, as it can be concluded from the presented analysis, none of the described multiagent middlewares provides the combination of features available in Siebog, namely the HTML5-based agent support on the client side, and the Java EE-based agent support for clustered environments on the server side. This combination of functionalities is in line with modern approaches to enterprise web application development, enabling an easier integration of Siebog and its agents into mainstream enterprise solutions.

## 6.2 Reasoning and cognitive architectures

Throughout the literature, concrete system architectures developed as part of the AGI research are referred to as *cognitive* or *reasoning*. Although the two terms denote similar things, there are some differences. As discussed in (Wang, 2006, 2013), reasoning is performed at a higher-level of abstraction and includes one or more cognitive functions, such as decision making and learning. It is not concerned with lower-level details, such as perceptual and motor skills, often

[6] `http://www.oracle.com/technetwork/java/index-jsp-142945.html`, retrieved on August 12, 2014.

[7] `http://www.recursionsw.com/products/voyager/voyager-intro.html`, retrieved on August 12, 2014.

found in cognitive architectures.

Cognitive architectures can generally be organized into three categories: *symbolic*, *emergent* or *connectionist*, and *hybrid* (Duch et al., 2008; Goertzel et al., 2010b; Oentaryo and Pasquier, 2008; Polk and Seifert, 2002). Symbolic architectures manipulate symbols at a higher level of abstraction, whereas the emergent architectures incorporate individual units for processing lower-level signals that flow through the network. Hybrid architectures represent combinations of the earlier two.

Obviously, DNARS is a symbolic reasoning architecture. This section first analyzes a number of well-established symbolic and hybrid reasoning and cognitive architectures. For additional information on these and other systems, see e.g. (Chong et al., 2007; Duch et al., 2008; Goertzel et al., 2010b; Oentaryo and Pasquier, 2008; Polk and Seifert, 2002; Thorisson and Helgasson, 2012). Afterwards, the focus will be on concrete BDI implementations, i.e. those used by the multiagent community.

### 6.2.1 Symbolic and hybrid architectures

*ACT-R* is a hybrid cognitive architecture, based on the so-called *Unified Theories of Cognition* (Newell, 1994), as well as the cognitive neuroscience research (Anderson et al., 2004; Duch et al., 2008; Lebiere and Anderson, 1993). For example, the ACT-R operation is based to a certain extent on the experimental data obtained from neuroimaging, such as *Functional Magnetic Reasoning Imaging* (fMRI) and by observing how different parts of the brain interact during the reasoning process. As such, ACT-R can also be used as a framework for emulating human reasoning.

The most important components of the ACT-R architecture include the *perceptual-motor* sub-system, for obtaining visual information about the world and performing physical actions, the *goal module*, which manages the system's intentions, and the *declarative module*, which holds the system's overall knowledge (Anderson et al., 2004). A limited amount of the information from each component is stored into corresponding *buffers*, to be used by the *central production system* for component coordination. Symbolic pieces of information (*chunks* in declarative, or *productions* in procedural knowledge) are also described by numerical parameters, allowing the construction of a *associative memory/network* (Duch et al., 2008; Lebiere and Anderson, 1993).

*ICARUS* is a symbolic cognitive architecture with several types of memories (Duch et al., 2008; Langley and Choi, 2006). Its *perceptual* memory includes descriptions of observed objects, the *belief* memory describes relations among objects, while the *conceptual* memory holds general knowledge. In each inference cycle, the system observes its environment and creates a set of percepts. The percepts are then matched against the conceptual knowledge to deduce new beliefs. Additional two memories are introduced to guide and control the system's behavior. *Goal* memory includes the system's actively managed goals, while the *skill* memory describes complex, hierarchical activities that the system can perform. It is worth noting that, among the cognitive architectures described

in this sub-section, the architecture of ICARUS bears the closes resemblance to the BDI agent architecture.

*OpenCog* represents a general-purpose framework for AGI researchers. It is not a cognitive architecture *per se* but a collection of reusable modules, which provide data structures and algorithms for building concrete cognitive architectures (Hart and Goertzel, 2008). Several systems have been built on OpenCog, including the *OpenCog Prime* and *OpenCogBot* cognitive architectures (Goertzel, 2009; Goertzel et al., 2010a).

*OpenNARS* is a reference open-source implementation[8] of non-axiomatic reasoning (Wang, 2006, 2013). The latest version implements the logic of all 9 layers of NAL as defined in (Wang, 2013). Its architecture consists of the memory module, the inference engine, and a *control* mechanism, which handles the system's reasoning cycles (Wang, 2006).

*Soar* is one of the earliest, and a well-known symbolic cognitive architecture (Duch et al., 2008; Laird, 2012; Laird et al., 2012). As ACT-R, it represents a concrete realization of the Unified Theories of Cognition. Soar programs are specified in the form of *if-then production rules*, which, in turn, are used to select and apply *operators* and execute actions. The system's knowledge is divided into the *long-term* and *working memory*. The long-term memory can be *procedural*, containing the knowledge on how to do things, *semantic*, containing declarative knowledge about the world, and *episodic*, which summarizes the previous experience.

The working memory of Soar contains knowledge that is relevant to the current situation, and is directly tied to the perception, action, and decision making modules. Several extensions of the core Soar architecture have been proposed as well, including the use of Reinforcement learning in operator selection, visual imagery modules, semantic and episodic learning, etc. (Duch et al., 2008; Laird, 2008).

OpenNARS and DNARS represent concrete realization of non-axiomatic reasoning. Their differences from other cognitive and reasoning architectures stem from the use of NAL as the underlying formalism. For example, no other system deals with the issue of insufficient knowledge and resources to the degree done in NAL. In addition, unlike many systems described here, OpenNARS and DNARS are more focused on emulating the human thought processes at a higher level of abstraction, rather than trying to accurately model the human brain (Wang, 2006). However, it remains to be seen which approach works the best, as all the systems are yet far from reaching the goal of building a "thinking machine."

DNARS is built by combining NAL and the Big Data paradigm, because the two try to solve a similar issue: how to handle and process large amounts of information with limited time and resources. NAL, for example, includes inference rules that deal with knowledge inconsistencies only when necessary, e.g. when there are different answers to the same question. It also includes constructs for combining individual pieces of information and reducing the amount of raw information. Similarly, the NoSQL database used in DNARS includes a number

[8] `https://github.com/opennars/opennars`, retrieved on August 12, 2014.

of techniques for dealing with large amount of information and strict time constraints, and temporarily sacrifice information consistency if needed. Therefore, in DNARS we combine the "best of both worlds" in an efficient manner.

There are some important differences between OpenNARS and DNARS. OpenNARS has been developed for a significantly longer period of time, and is a more mature product. In the latest version, OpenNARS implements all layers of NAL, and includes more advanced control mechanisms. The main advantage of DNARS, however, is in the organization of its backend knowledge base. That is, DNARS is currently capable of reasoning over much larger knowledge bases than OpenNARS. By utilizing modern approaches to large-scale data processing, DNARS can easily be used to, for example, realize the case study presented in Section 5.4.

### 6.2.2 Concrete BDI implementations

As noted, BDI is to most popular model for developing intelligent agents. Over time, several interesting concrete realization of the model has been proposed (Bordini et al., 2007; Braubach et al., 2013; D'Inverno et al., 2004; Georgeff and Lansky, 1987; Hindriks, 2014; Jarvis et al., 2010; Nunes et al., 2011).

*BDI4JADE* extends JADE with the support for BDI agents (Nunes et al., 2011). Its authors argue that, although sometimes convenient, agent-oriented programming languages usually represent a barrier that limits the wider adoption of the BDI model. Therefore, the BDI4JADE framework is based on pure Java.

BDI4JADE agents are defined through their *capabilities*, which include plans and relevant beliefs along with public interfaces. Additional essential components include desires, intentions and goals, with their usual meanings, *events* that signal changes in the goal and belief bases, as well as *strategies* for customizing the reasoning cycles. A reasoning cycle includes a number of steps (Nunes et al., 2011), which can be summarized as follows. The agent first revises its belief base, removes completed goals, and then proceeds to choosing a set of *applicable* goals (i.e. desires). A subset of desires is selected for achievement becoming the agent's intentions. Finally, active intentions are associated with plans that can fulfill them.

*Procedural Reasoning System* (PRS) is one of the earliest agent architectures based on the BDI model (Georgeff and Lansky, 1987). It includes four databases, containing agent's beliefs, goals, declarative procedures (i.e. plans), and intentions (i.e. active plans). These databases are managed the *interpreter*, which operates in reasoning cycles. In each cycle it selects *applicable* plans, whose pre-conditions match the current beliefs and goals. One applicable plan is then selected, placed on the intention stack, and then executed. During the execution, new beliefs and/or goals may be generated, which will create new intentions. Finally, it is worth noting that multiple PRS interpreters can operate in parallel and communicate with each other.

More recently, PRS has been extended in form of the *Distributed Multi-Agent Reasoning System* (dMARS) (D'Inverno et al., 2004). In addition to beliefs, goals, plans, and intentions, dMARS supports external and internal ac-

tions, which, respectively, affect the environment or the agent's state. A plan can include the total of six components: a triggering event, pre-conditions, a body, a *maintenance condition* which must hold throughout the plan's execution, and two sets of internal actions to be executed if the plain succeeds or fails. dMARS has reportedly seen some important practical applications, including NASA space shuttle fault diagnosis, air traffic control, supply chain management, etc. (Mascardi et al., 2005)

*GOAL* is a practical agent-oriented programming language (Hindriks, 2014, 2009). The *mental state* of a GOAL agent is defined through a static knowledge base, a dynamic belief base, as well as different types of goals. Active goals are removed from the agent's mental state using the *blind commitment strategy*, which means that only successfully achieved goals are dropped (Hindriks, 2014, 2009; Rao and Georgeff, 1993). The action execution strategy is guided by so-called *action rules*. They are specified in the form of *IF mental_state THEN action*. If the given mental state is true, the action is said to be *applicable*. An action that is both applicable and enabled (Sub-section 1.4.4) is called an *option*. Action rules can be checked in several way (e.g. in the order they are written, randomly, etc.), and the first action that becomes an option is executed.

*GORITE* is a BDI framework that highlights teamwork as the main advantage of the agent technology over other A(G)I approaches (Jarvis et al., 2010; R onnquist, 2008). Therefore, a team of agents is viewed as a distinct entity, with its own beliefs, desires, intentions, and goals. Each team member is assigned one or more *roles*, where a role is defined as a set of related goals (Jarvis et al., 2010). When a goal needs to be achieved by the team, a subset of agents with the required roles is selected and activated. The default sub-team selection process can be customized by the end-user.

*Jadex* follows the object-oriented model for representing beliefs and goals, instead of the more common approach based on logical formulae (Braubach et al., 2013; Pokahr and Braubach, 2008). The Jadex infrastructure consist of the agent platform (e.g. standalone or JADE), *active components*, and *kernels*. Active components, broadly speaking, represent the merger of agents and service-oriented systems, while kernels define internal workings of active components. Here, the most important is the *BDI agent kernel* (Braubach et al., 2013). It is based on the PRS described earlier, with the addition of the *goal deliberation* technique for maintaining a consistent set of goals.

It is worth noting that, among the BDI architectures described here, Jadex is currently the most actively developed system. In the latest version, Jadex agents can be written using pure Java (Pokahr et al., 2014). Different BDI elements can be specified using annotations, which is also the approach used in Siebog-DNARS integration.

*Jason* is a popular interpreter for an agent-oriented programming language *AgentSpeak* and a reasoning engine for BDI agents (Bordini and Hubner, 2006; Bordini et al., 2007). Agents are defined in terms of beliefs, goals, and plans. The interpreter operates in reasoning cycles, divided into 10 individual steps. First, the agent perceives its environment (generating a perceptual information), pro-

cesses a single message received from another agent, while filtering-out "socially unacceptable" messages, and updates its belief base accordingly. The remaining six steps represent the core of agent's reasoning and acting:

- A single *event* is selected to be processed. An event represents a change in the agent's mental state (e.g. a new belief has been added).
- A set of *relevant* plans, i.e. plans corresponding to the selected event, is constructed.
- Of those, a set of *applicable* plans (also called *options*) is determined.
- An applicable plan is put on a stack to become an *intention*. This is the plan to which the agent will commit.
- An intention is selected from the stack.
- A single step of the selected intention is executed.

Jason is designed as a highly-customizable architecture, and has been integrated with a number of other agent-based systems[9].

NAL provides a number of advantages over the traditional BDI model. First and foremost, NAL statements are associated with truth-values. In concrete BDI implementations discussed earlier, there is no way of expressing the agent's confidence in a belief; it is left to the agent developer to somehow handle the notion that a belief might not be true. NAL statements, on the other hand, are *beliefs* in their true definition.

Additionally, unlike the BDI model, inconsistency resolutions (through backward inference), learning (through forward inference), and working under the assumption of insufficient knowledge and resources (e.g. compound terms discussed in Section 2.5), represent inherent features of NAL-based agents.

These are the main advantages of DNARS over the presented BDI systems. Finally, as in OpenNARS-DNARS comparison, DNARS offers the possibility of reasoning over much larger knowledge bases than any existing BDI system. This opens up DNARS to a wider range of possible practical applications, as demonstrated by the case study in Section 5.4.

## 6.3 Summary

Obviously, the work presented in this thesis belongs to the thriving scientific areas. That is, multiagent middlewares, BDI agent architectures, as well as general-purpose cognitive and reasoning systems, have received a great deal of attentions from artificial intelligence and artificial general intelligence (AGI) researchers and practitioners.

[9] `http://jason.sourceforge.net`, retrieved on August 12, 2014.

When compared to existing systems, the Siebog multiagent middleware offers numerous advantages, which stem from the use of standard software development techniques and principles. To recall, these include scalability and fault-tolerance on the server, true platform independence on the client, as well as cross-platform interaction, code sharing, and heterogeneous mobility. No other existing multiagent middleware provides the benefits of both client-side and server-side technologies to the degree achieved in Siebog.

As shown, many AGI researches and practitioners have moved away from BDI as the model for developing intelligent software systems. This is also the approach taken in Siebog: instead of the BDI model, intelligent agents in Siebog rely on the non-axiomatic logic realized in form of a distributed system.

The next, and final, chapter of this thesis summarizes the completed work, and also proposes the planned future course of development.

# Chapter 7

# Conclusions and Future Work

Software agents represent one of the most consistent approaches to distributed artificial intelligence, and distributed computing in general. Agents are, first and foremost, social (artificial) entities. This allows them to share the workload, cooperate and coordinate their actions, and even negotiate and compete against each other in order to fulfill the target goals.

Agents need a run-time environment that supports their execution. The job of this multiagent middleware is to provide, among other things, efficient and reliable communication channels, a yellow-pages service, mobility, and execution in distributed environments. A large part of the work presented in this thesis has been focused on designing a novel multiagent middleware that provides these, and other functionalities, but which also takes into an account the functional requirements of modern enterprise and web applications.

As discussed thoroughly in Chapter 1, the two prevalent definitions of agents include the so-called weak and strong notions of agency. According to the weak notion, the main characteristics of agents include autonomous, reactive, proactive, and social behavior. The work presented in this thesis, however, is concerned more with the strong notion, which also includes the concepts of intelligent and human-like behavior.

## 7.1 The work done

The completed work and the main contributions of the thesis can be summarized as follows.

First, Chapter 3 proposes an architecture of a new system for intelligent agents. It discards the BDI model commonly used by the agent research community, and instead it aligns with the AGI research community. The proposed system, named *Distributed Non-Axiomatic Reasoning Systems* (DNARS), uses the *Non-Axiomatic Logic* (NAL) as its formal reasoning framework.

As discussed in Chapter 2, NAL provides a well-defined syntax, experience-grounded semantics, and a set of inference rules, but works under the *assumption of insufficient knowledge and resources* (Wang, 2013). The main novelty of DNARS, especially when compared to OpenNARS, is its ability to efficiently handle large quantities of knowledge, while providing service to high numbers of external clients. This ability was achieved by a uniquely designed backend knowledge base, and a set of algorithms that adequately realize NAL inference rules in these distributed, highly-scalable settings.

The second main contribution of the thesis, presented in Chapter 4, is the new Siebog multiagent middleware. Although many multiagent middlewares already exist, none of them provides the full set of advantages offered by Siebog. For example, the client-side of Siebog is executed in web browsers, without any external requirements. Therefore, it is supported on a wide variety of hardware and software platforms, such as desktop computers, smartphone and tablet devices, Smart TVs, etc. On the server side, Siebog uses modern enterprise software development standards, in order to provide high-availability of its agents, namely through load-balancing, and state replication and failover.

However, the Siebog is more than a "sum of its parts." Unlike existing agent middlewares, it combines web and enterprise technologies into a unified framework which allows for agent code sharing, heterogeneous agent mobility, and cross-platform communication.

Finally, Siebog has been extended with the support for DNARS-based intelligent agents. The final result of this work is a multiagent middleware with a unique architecture, and a unique reasoning system for intelligent agents. It offers new and interesting ways of applying the agent technology in a range of domains. An example of extending the *DBpedia* project by deriving new structured knowledge has been shown in Section 5.4, while other possibilities for practical application include intelligent virtual assistants (overview of agents in knowledge management, 2006), systems with large agent societies (Ilie et al., 2011), the domains of *Internet of Things* and smart environments (Nakashima et al., 2010), etc.

As indicated earlier, this work is a long-term, ongoing research effort. The completed work opens up an array of new questions, problems and possibilities for further research. Some of the planned future research directions are discussed in the next section.

## 7.2 Open questions and opportunities for future work

The two main results of the thesis, Siebog and DNARS, offer several possibilities for future research directions. Both as separate systems and parts of the unified framework, they pose a number of challenges that need to be solved.

Obviously, the remaining layers of NAL need to be added to DNARS. As shown in Section 5.4, currently implemented NAL layers are sufficient for simple

reasoning tasks. The remaining layers, however, would provide agents with higher-level reasoning capabilities. In particular (Wang, 2013):

- Starting from layer NAL-5, statements can be used as terms, and new copulas (such as *implication* and *equivalence*) are supported.
- NAL-6 adds support for variables, and would enable agents to work with more general rules.
- NAL-7 introduces the concepts of time and events, as well as temporal connectors (e.g. *sequential* and *parallel*) and relations (e.g. *before* and *when*).
- Procedural knowledge, in form of operations and goals, is added in NAL-8.
- Finally, agents based on NAL-9 would be capable of processing emotions, and exhibit self-monitoring and self-control.

Regarding the Siebog middleware, several directions of improvements are planned as well. For agent developers who insist on using the BDI model, we plan to integrate Jason into the server-side of Siebog. This would enable AgentSpeak/Jason agent to run in Java EE environments, and employ automatic load-balancing and fault-tolerance.

An implementation of a higher-level, agent-oriented programming language for Siebog, inspired by ALAS (Mitrović et al., 2012a), is planned as well. This would bring the existing code sharing feature of Siebog to a new level, allowing developers to write the agent code only once.

Finally, although fully functional on the server, the Siebog-DNARS integration needs to be further developed on the client side in order to allow direct representations of NAL concepts. This could be achieved directly in JavaScript, or via the proposed agent-oriented programming language.

# Prošireni izvod

Disertacija se sastoji iz 7 glava, podeljenih u tri dela. Deo I definiše osnovne pojmove. Glava 1 predstavlja opis generalnih principa i koncepata softverskih agenata i agentske tehnologije uopšte, koji predstavljaju osnovnu temu disertacije. Glava 2 uvodi osnove tzv. *Ne-Aksiomatske Logike* (NAL) koja predstavlja formalni okvir za rezonovanje korišćen u radu.

Deo II predstavlja glavne doprinose disertacije. U Glavi 3 je predstavljena arhitektura *Distribuiranog Sistema za Ne-Aksiomatsko Rasuđivanje* (eng. *Distributed Non-Axiomatic Reasoning System*) (DNARS). DNARS je zasnovan na principima NAL-a, u kombinaciji sa savremenim pristupima i standardima za obradu velikih količina podataka. Njegova jedinstvena arhitektura mu omogućuje da radi sa veoma velikim bazama znanja, te da odgovara na pitanja korisnika u realnom vremenu. Pored arhitekture, za potrebe DNARS-a su razvijeni i odgovarajući algoritmi koji mu omogućuju da efikasno izvršava zadate operacije.

Glava 4 opisuje multiagentsku platformu *Siebog*[1] koja kombinuje savremene principe serverskog i klijentskog razvoja sistema u jedinstven programski okvir za agente. Sa serverske strane, Siebog nudi automatsko raspoređivanje (eng. *load-balancing*) agenata po čvorovima klastera, kao i otpornost na hardverske i softverske greške (eng. *fault-tolerance*). Sa klijentske strane, Siebog funkcioniše kao platformski-nezavistan sistem, odnosno dizajniran je tako da može da se izvršava na velikom broju uređaja, poput klasičnih desktop računara, "pametnih" telefona i tableta, "pametnih" televizora, itd. Serverska i klijentska strana su, dalje, integrisane u jedinstveni okvir koji donosi višeplatformsku komunikaciju agenata, heterogenu mobilnost, kao i deljenje koda. Konačno, Siebog je povezan sa DNARS-om, kako bi se omogućio razvoj inteligentnih multiagentskih sistema sa jedinstvenim mogućnostima.

Tri konkretna primera praktičnih primena multiagentskih sistema zasnovanih na DNARS-u i Siebog-u su data u Glavi 5. Prvi primer demonstrira kako se pomoću Sieboga može razviti sistem čije funkcionalnosti odgovaraju postojećim mrežama za razmenu sadržaja. Drugi primer predstavlja rezultate eksperimenta koji pokazuju kako DNARS može davati odgovore na veliki broj pitanja, i to u

[1] U staroslovenskoj mitologiji, Siebog je bio bog ljubavi i braka. Naziv tako oslikava činjenicu da je predložena platforma nastala spajanjem dva postojeća sistema - XJAF i Radigost. Sa druge strane, Radigost je bio bog gostoprimstva (*radi*, odnosno *dragi gost*), što govori da su Radigost agenti dragi gosti klijentskih uređaja.

realnom vremenu. Treći i poslednji primer pokazuje kako se inteligentni multi-agentski sistem zasnovan na Siebog-u u DNARS-u može iskoristiti za praktičnu primenu generisanja novog struktuiranog znanja.

Konačno, Deo III opisuje relevantna postojeća istraživanja, poredi ih sa rezultatima disertacije (Glava 6) i, na kraju, sumira rezultate i predlaže mogućnosti za dalja istraživanja (Glava 7).

# Osnovni pojmovi i definicije

## Agenti

Iako ne postoji opšte prihvaćena definicija *softverskih agenata* (ili, jednostavno *agenata*), isti se mogu opisati kao *autonomni* sofverski entiteti, sa različitim nivoima *inteligencije*, koji su sposobni da deluju *samostalno* kako bi postigli zadate ciljeve (Wooldridge, 1999). Nedostatak sveobuhvatne definicije agenata potiče iz njihovih brojnih praktičnih primena; na primer, dok se neki problemi mogu uspešno rešiti upotrebom *mobilnih* agenata (Medvidovic and Edwards, 2010; Urra et al., 2010), za druge mobilnost samo uvodi nepotreban nivo kompleksnosti (Carzaniga et al., 2007).

Osnovna tema doktorske disertacije su inteligenti agenti, tj. agenti koji ispoljavaju određeni nivo inteligencije u vidu fleksibilnog, adaptivnog delovanja. Formalnije, u disertaciji se pod pojmom *inteligentni agent* podrazumeva agent definisan na sledeći način:

> "Inteligentnog agenta ili agentsku arhitekturu definišemo tako da sadrži eksplicitnu, simboličku reprezentaciju sveta i u kojoj se odluke (npr. o tome koje akcije primeniti) određuju na osnovu logičkog (ili bar pseudo-logičkog) rasuđivanja zasnovanog na prepoznavanju obrazaca i manipulaciji simbola". (str. 130 Wooldridge and Jennings, 1995)

Tokom godina je predloženo nekoliko internih arhitektura agenata. *Reaktivni agenti* su agenti koji kontinualno prilagođavaju svoje ponašanje promenama u svom okruženju (Bordini et al., 2007; Salamon, 2011; Wooldridge, 1999). Kao takvi, pogodni su za dinamična okruženja. *BDI* arhitektura (eng. *Belief-Desire-Intention*), sa druge strane, koristi koncepte verovanja, želja i namera kako bi opisala funkcionisanje agenta (Rao and Georgeff, 1995). Verovanja su činjenice za koje agent pretpostavlja da su tačne, iako to zapravo ne mora biti slučaj. Želje opisuju stanja sveta koje bi agent hteo da postigne, dok su namere želje kojima se agent posvetio. BDI arhitektura predstavlja najpopularniju i najviše istraženu arhitekturu inteligentnih agenata, sa brojnim teorijskim osnovama (Cohen and Levesque, 1990; Dunin-Keplicz and Verbrugge, 2010; Guerra-Hernandez et al., 2009; Singh et al., 1999; van der Hoek and Wooldridge, 2013) i praktičnim realizacijama (Bordini et al., 2007; Braubach et al., 2013; D'Inverno et al., 2004; Georgeff and Lansky, 1987; Hindriks, 2014; Jarvis et al., 2010; Nunes et al., 2011).

Agenti veoma retko funkcionišu samostalno, već rade u grupama, odnosno u *agentskim društvima*. Zapravo, ovaj socijalni aspekt je jedan od najbitnijih karakteristika agenata, odnosno karakteristika koja izdvaja agente od drugih grana veštačke inteligencije. Unutar grupe, agentska interakcija može imati različite forme, kao što su: direktna razmena poruka, koordinacija akcija, kooperacija i razmena znanja i iskustava, pregovaranje u cilju prevazilaženja konflikata ili, u slučaju koristoljubivih agenata, tačkmičenje i sabotaža (Huhns and Stephens, 1999; Salamon, 2011). Interakcija socijalnih agenata je temeljno istražena naučna oblast, koja je rezultovala brojnim standardima i protokolima, kao što su *Contract Net* (FIPA CNet), *sistem table* (eng. *the blackboard system*) (Corkill, 2003; Huhns and Stephens, 1999), *kooperativno učenje* (Panait and Luke, 2005; Sen and Weiss, 1999), *inteligencija roja* (eng. *swarm intelligence*) (Blum and Merkle, 2008; Panigrahi et al., 2011), itd.

Kako bi uspešno obavljali svoje zadatke, agentima je neophodna odgovarajuća arhitektura. Ove arhitekture, poznate pod nazivom *multiagentski programski okviri* ili *multiagentske platforme* (eng. *multiagent framework* ili *multiagent platform*, još i *multiagent middleware*), uključuju efikasnu infrastrukturu za agentsku komunikaciju, obezbeđuju podršku za mobilnost agenata, omogućuju agentima da objavljuju svoje mogućnosti i da pretražuju mogućnosti drugih agenata, itd.

Formalnije, funkcionalnosti multiagentskog programskog okvira su standardizovane u okviru specifikacije za upravljanje agentima koju je donela *Fondacija za inteligentne fizičke agente* (eng. *Foundation for Intelligent Physical Agents*) (FIPA) (FIPA Home). Ova FIPA specifikacija definiše tri osnovne komponente multiagentskog okvira:

- *Sistem za upravljanje agentima* (eng. *Agent Management System*, AMS) (FIPA Ams), zadužen za registraciju i deregistraciju agenata, kao i pretragu registrovanih agenata.
- *Moderator direktorijuma* (eng. *Directory Facilitator*, DF) (FIPA Ams), koji ima ulogu "žutih strana", odnosno objavu i pretraživanje mogućnosti agenata.
- *Servis za transport poruka* (eng. *Message Transport Service*, MTS) (FIPA Mts), čiji je zadatak prenos poruka između agenata.

Specifikacija još definše AMS i MTS kao obavezne komponente konkretne multiagentske platforme, dok je DF opciona komponenta. Primer unutrašnje organizacije agentske platforme i odnos komponenti je dat na Slici 1.5.

Tokom godina je razvijen veliki broj konkretnih multiagentskih platformi, ali se, nažalost, relativno malih broj njih i dalje aktivno razvija (Bădică et al., 2011). Neke od popularnijih i uticajnijih platformi su opisane u Poglavlju 1.3, kao i u Glavi 6.

## Ne-Aksiomatska Logika

*Ne-Aksiomatska Logika* (eng. *Non-Axiomatic Logic*) (NAL) je formalizam za specifikaciju sistema za rezonovanje u okviru *Veštačke opšte inteligencije* (eng.

*Artificial General Intelligence*) (Wang, 2006, 2013; Wang and Awan, 2011). NAL obuhvata gramatiku, odnosno alfabet, skup pravila za izvođenje, i semantičku teoriju. Za razliku od mnogih drugih formalizama koji se koriste u računarstvu, međutim, NAL je *logika termova* (Smith, 2012; Sommers and Englebretsen, 2000; Wang, 2013): rečenice su date u obliku *subjekat-relacija-objekat*, pri čemu su subjekat i objekat termovi.

Pojam *ne-aksiomatska* označava da je logika pogodna za razvoj sistema koji funkcionišu u uslovima *nedovoljnog znanja i resursa* (Wang, 2013; Wang and Awan, 2011). To najpre znači da je znanje koje sistem poseduje nesigurno i ne obavezno konzistentno. Novi dokazi se mogu pojaviti u bilo kom momentu, mogu imati bilo kakav sadržaj i mogu promeniti istinitost bilo koje postojeće rečenice. Pored toga, sistem najčešće nema dovoljno resursa (u smislu vremena, memorijskog prostora, itd.) da konsultuje svoje celokupno znanje kako bi rešio problem. Dodatno, sistem ne može primeniti pun skup pravila za izvođenje niti prati neki unapred zadati algoritam.

NAL uključuje brojne mehanizme za rad u uslovima nedovoljnog znanja i resursa. Na primer, može efikasno upravljati nekonzistentnim bazama znanja, te sumirati postojeće znanje i time umanjiti ukupan broj rečenica u bazi.

Čitava logika je organizovana u devet nivoa. Svaki nivo uvodi dodatnu gramatiku i pravila izvođenja, te proširuje ekspresivnost logike i mogućnosti sistema koji su na njoj zasnovani. Prvi nivo, u oznaci NAL-1, definiše osnovnu relaciju, *nasleđivanje*.

**Definicija 7.1.** *Rečenica nasleđivanja* je rečenica oblika $S \to P$, gde su $S$ i $P$ termovi koji označavaju, redom, subjekat i objekat, dok $\to$ označava relaciju nasleđivanja (Wang, 1994, 2006, 2013).

Nasleđivanje je po definiciji tranzitivno i refleksivno, pri čemu se rečenice oblika $S \to S$ nazivaju *tautologije* i najčešće se ne uključuju u bazu znanja sistema. Rečenica nasleđivanja $S \to P$ neformalno označava *$S$ je tipa $P$*, npr. *mačka je tip životinje*. U prvom nivou, subjekat i predikat su atomski termovi (tj. reči). Na višim nivoima, termovi se mogu sastojati od više reči povezanih konektorom, te mogu biti i cele rečenice.

NAL koristi tzv. *semantiku zasnovanu na iskustvu* (eng. *experience-grounded semantic*) (Rodriguez and Geldart, 2009; Wang, 2005). Po ovoj semantici, term $S$ ima značenje za sistem samo ako postoji u prethodnom iskustvu sistema. Na osnovu relacije terma $S$ sa drugim termovima utvrđuju se količine *pozitivnih* i *negativnih dokaza*, na osnovu kojih se, dalje, izvode istinitosne vrednosti rečenica.

**Definicija 7.2.** Istinitosna vrednost NAL rečencice je par realnih brojeva u intervalu $[0, 1]$, koji se nazivaju *učestalost* ($f$) i *poverenje* ($c$). Učestalost predstavlja odnos pozitivnih i ukupnih dokaza, dok poverenje definiše koliko će učestalost biti stabilna kada se pojave novi dokazi. (Wang, 1994, 2001b, 2006, 2013).

Pravila izvođenja u NAL-u imaju *silogistički oblik* (Smith, 2012). Silogističko pravilo izvođenja prihvata dve rečenice sa deljenim termom i izvodi novi za-

ključak koji obuhvata preostale termove. Pravila izvođenja u NAL-u su organizovana u tri grupe (Wang, 2006, 2013): *izvođenja unapred* (eng. *forward inference*) izvode nove zaključke, *izvođenja unazad* (eng. *backward inference*) odgovaraju na pitanja, dok *lokalna izvođenja* (eng. *local inference*) rade sa nekonzistentnim rečenicama.

Tri osnovna pravila izvođenja unapred u NAL-1 su *dedukcija*, *indukcija* i *abdukcija*, koje se, redom, definišu na sledeći način (Wang, 2001a, 2013):

$$\{M \to P\langle f_1, c_1\rangle, S \to M\langle f_2, c_2\rangle\} \vdash S \to P\langle f_1 f_2, f_1 f_2 c_1 c_2\rangle \tag{7.1}$$

$$\{M \to P\langle f_1, c_1\rangle, M \to S\langle f_2, c_2\rangle\} \vdash S \to P\langle f_1, \frac{f_2 c_1 c_2}{f_2 c_1 c_2 + k}\rangle \tag{7.2}$$

$$\{P \to M\langle f_1, c_1\rangle, S \to M\langle f_2, c_2\rangle\} \vdash S \to P\langle f_2, \frac{f_1 c_1 c_2}{f_2 c_1 c_2 + k}\rangle \tag{7.3}$$

Sledeći nivo, NAL-2, uvodi novu relaciju – *sličnost* – i odgovarajući skup novih pravila izvođenja.

**Definicija 7.3.** Sličnost, u oznaci $\leftrightarrow$ , je relacija koja se definiše kao simetrično nasleđivanje: $(S \leftrightarrow P) \Leftrightarrow (S \to P) \wedge (P \to S)$ (Wang, 2009, 2013).

Rečenica sličnosti $S \leftrightarrow P$ neformalno označava *S je P*, npr. *tigar je mačka*. Kao kod nasleđivanja, rečenice oblika $S \leftrightarrow P$ su tautologije, koje se najčešće ne uključuju u bazu znanja sistema.

Relacija sličnosti omogućuje uvođenje tri nova pravila izvođenja (Wang, 2006, 2009, 2013): *poređenje*, *analogiju* i *nalikovanje* (eng. *resemblance*). Poređenje postoji u dve varijante, koje su poput indukcije odnosno abdukcije, pri čemu je jedna od premisa rečenica sličnosti. Analogija je poput dedukcije, pri čemu je, ponovo, jedna od premisa rečenica sličnosti. Konačno, nalikovanje prihvata dve rečenice sličnosti kao premise i izvodi zaključak koji je takođe rečenica sličnosti.

Nivo NAL-3 uvodi mogućnost rada sa složenim termovima. Složeni term se sastoji od jednog ili više atomskih ili složenih termova, koji su povezani odgovarajućim konektorom.

**Definicija 7.4.** Složeni term je term oblika $\{T_1\ con\ T_2\ con \ldots con\ T_n\}$, gde *con* predstavlja konektor[2], a $T_1 \ldots T_n$ su atomski ili složeni termovi, $n \geq 1$ (Wang, 2006, 2013).

NAL-3 podržava četiri tipa standardnih konektora: *proširujući presek* ($\cap$), *sužavajući presek* ($\cup$), *proširujuća razlika* ($-$), i *sužavajuća razlika* ($\ominus$), definisanih u jednačinama 2.26 do 2.29. Konačno, uvedeno je i novo pravilo izvođenja, *kompozicija*, koje koristi ova četiri konektora kako bi sumiralo postojeće znanje u bazi i time efektivno smanjilo broj rečenica sa kojima sistem mora da radi.

Kako bi podržao proizvoljne relacije između termova, nivo NAL-4 najpre uvodi novi konektor, *proizvod*, u oznaci $\times$. Ovaj konektor definiše odnos između komponenti složenih termova:

[2]Podržana je i infiksna notacija, tj. $\{con\ T_1\ T_2\ \ldots\ T_n\}$

$$((S_1 \times \cdots \times S_n) \rightarrow (P_1 \times \ldots \times P_n)) \Leftrightarrow ((S_1 \rightarrow P_1) \wedge \ldots \wedge (S_n \rightarrow P_n)) \quad (7.4)$$

**Definicija 7.5.** *Relacioni term* $R$ se definiše kao atomski term koji je sa složenim proizvodnim termom povezan relacijom nasleđivanja, odnosno:
$(T_1 \times T_2) \rightarrow R$ ili $R \rightarrow (T_1 \times T_2)$ (Wang, 2006, 2013).

NAL-5 uvodi rečenice višeg reda, odnosno rečenice čiji su termovi rečenice. NAL-6 donosi podršku za promenljive, NAL-7 uvodi pojam vremena, dok NAL-8 uključuje podršku za operacije, odnosno proceduralne rečenice. Konačno, NAL-9 definiše samokontrolu i samo-praćenje. Disertacija se, međutim, detaljnije bavi samo nivoima NAL-1 do NAL-4. Kako je pokazano u Glavi 5, sistem za rasuđivanje zasnovan na prva četiri nivoa NAL-a je dovoljan za razvoj inteligentnih agenata sa konkretnim, praktičnim primenama.

# Rezultati disertacije

## DNARS

Za sistem u domenu veštačke inteligencije, mogućnost rada sa velikim bazama znanja predstavlja jednu od ključnih osobina (Hovy et al., 2013). *Distribuirani sistem za ne-aksiomatsko rasuđivanje* (eng. *Distributed Non-Axiomatic Reasoning System*) (DNARS) je sistem za rasuđivanje koji je zasnovan na NAL-u i generalnim principima razvoja ne-aksiomatskih sistema za rasuđivanje (Wang, 2006, 2013). Novine koje DNARS donosi su jedinstvena arhitektura i odgovarajući skup algoritama koji mu upravo omogućuju da radi sa veoma velikim bazama znanja.

NAL se koristi kao formalizam u DNARS-u, jer se "filozofija" iza ove logike uklapa u opšte ciljeve DNARS-a. Kao što je ranije rečeno, NAL je izgrađen oko koncepta nedovoljnog znanja i resura. Na primer, za razliku od mnogih formalizama koji se koriste u agentskoj tehnologiji (npr. Guerra-Hernandez et al., 2009; Mora et al., 1999; Singh et al., 1999), NAL može da radi sa nekonzistentnim bazama znanja, te da se oslanja na pravila *revizije* i *izbora* kako bi rešio ove probleme (Wang, 2013). S tim u vezi, sama arhitektura DNARS-a i tehnologije koje su korišćene za njenu realizaciju takođe žrtvuju konzistentnost podataka u cilju skalabilnosti (Gilbert and Lynch, 2002; Hewitt, 2010).

Arhitektura DNARS-a je predstavljena na Slici 3.1. Osnovne komponente predloženog sistema su:

- *Sistem za odlučivanje*: odgovara na pitanja korisnika.
- *Sistem za izvođenje unapred*: izvodi nove zaključke i znanja.
- *Kratkotrajna memorija*: sadrži rečenice koje su od značaja za trenutni ciklus rasuđivanja i probleme koje treba rešiti.

- *Oblast znanja*: podskup celokupne baze znanja koji sadrži međusobno zavisne ili povezane rečenice.
- *Pozadinska baza znanja*: celokupno znanje i iskustvo sistema.
- *Rukovodilac događajima*: upravlja događajima koji opisuju promene u bazi znanja.

Prilikom definisanja arhitekture DNARS-a, posebna pažnja je posvećena organizaciji baze znanja. Baza je dizajnirana u vidu distribuirane, skalabilne arhitekture koja se sastoji iz tri nivoa. Na najnižem nivou, celokupno znanje je izdeljeno i podeljeno na više računara u klasteru. Upotrebljeno je tzv. *horizontalno skaliranje*: kako se količina znanja povećava, performanse sistema se održavaju jednostavnim dodavanjem novih računara u klaster (Michael et al., 2007). Ovaj pristup ima dve glavne prednosti:

- Baza znanja može da sadrži i upravlja sa velikim količinama podataka. Uvođenjem odgovarajućih pravila za distribuciju podataka, omogućeno je brže pronalaženje i pribavljanje relevantnih rečenica.
- Omogućena je otpornost sistema na greške, jer se podaci kopiraju na računare klastera; tj. ne postoji *jedna tačka neuspeha* i sistem može nastaviti da funkioniše bez obzira na hardverske ili softverske greške.

Na drugom, srednjem nivou, celokupno znanje je podeljeno na jednu ili više *Oblasti znanja*. Oblasti organizuju podatke iz nižeg nivoa u logičke kategorije i omogućuju mu da radi sa podskupom znanja. Tokom izvršavanja, sistem može konsultovati jednu ili više oblasti. Ova organizacija takođe podržava višeklijentsku prirodu DNARS-a. Konkretno, znanje koje pripada jednom klijentu može biti smešteno u jednu oblast. Ali, više klijenata može raditi sa istom oblašću (jednom ili više) i time deliti znanje i stečeno iskustvo.

Konačno, *Kratkotrajna memorija* se nalazi na trećem, najvišem nivou apstrakcije. Sadrži znanje koje je direktno dostupno pravilima za izvođenje, te predstavlja polaznu osnovu za rasuđivanje u DNARS-u. Glavni zadatak Kratkotrajne memorije je da služi kao modul za optimizaciju izvršavanja. Njen sadržaj bi trebalo u potpunosti da stane u radnu memoriju jedne mašine, pružajući maksimalne performanse. Kada se skup relevantnih ciklusa rasuđivanja završi, sadržaj Kratkotrajne memorije se upisuje u i spaja sa odgovarajućim Oblastima znanja (jednom ili više).

Za klijente je veoma važno da budu obavešteni o promenama u bazi znanja, naročito ako više klijenata deli istu Oblast znanja. Na primer, klijent može želeti da preduzme niz akcija kao odgovor na novostečeno znanje. Da bi podržao ove funkcionalnosti, DNARS uključuje Rukovodioca događajima. Promena u Oblasti znanja može generasati jedan ili više događaja, koji će zatim biti prikupljeni od strane Rukovodioca i poslati odgovarajućim klijentima. Na ovaj način, DNARS nudi mogućnost razvoja reaktivni inteligentnih agenata (Wooldridge and Jennings, 1995).

Nakon predstavljanja opšte specifikacije DNARS-a, njegove arhitekture i osnovnih funkcionalnosti u Poglavlju 3.2, predložena su i dva moguća pristupa za konkretnu realizaciju sistema.

Baza znanja zasnovana na NAL rečenicama se zapravo može predstaviti pomoću tzv. *grafa osobina* (eng. *property graph*). Graf osobina je usmereni graf za različitim tipovima grana i sa, dodatno, osobinama "zakačenim" za čvorove i grane (Robinson et al., 2013; Rodriguez and Shinavier, 2010). U ovakvog grafu, termovi NAL rečenica su predstavljeni čvorovima, relacije (nasleđivanje i sličnost) su predstavljane odgovarajućim granama, dok su istinitosne vrednosti rečenica osobine zakačene za ove grane.

Kada se baza znanja predstavi grafom osobina, pravila izvođenja se mogu realizovati u vidu algoritama za pretraživanje i procesiranje grafova, kao što je opisano u Poglavlju 3.5. Na primer, neka je DNARS-u dat zadatak da pronađe najbolji odgovor na pitanje oblika $S \rightarrow ?$. Odnosn, potrebno je pronaći term koji nedostaje, tako da odgovarajuća rečenica ima najbolju istinitosnu vrednost (Poglavlje 2.8). U ovoj situaciji, DNARS prikuplja sve grane koje napuštaju čvor $S$, sortira ih po odgovarajućim kriterijumima i vraća ulazni čvor najbolje grane.

Kao alternativno rešenje, u Poglavlju 3.7 je iskorišćen model programiranja poznat pod nazivom *MapReduce* (Dean and Ghemawat, 2008; White, 2012). Predstavljeni su algoritmi koji funkcionišu u strogo distribuiranim sistemima, deleći bazu znanja na *kolekcije* i realizujući pravila izvođenja u obliku funkcija nad tim kolekcijama. Na taj način je velike količine podataka moguće veoma efikasno obraditi u skoro realnom vremenu, kao što zahteva data specifikacija DNARS-a.

Problem je, međutim, u tome što je realizacija zasnovana na *MapReduce* modelu komplikovana i često teško razumljiva. Time je, dugoročno gledano, teška za održavanje i dodavanje novih funkcionalnosti iz npr. viših nivoa NAL-a. S toga DNARS u osnovni podešavanjima koristi algoritme za izvođenje koji su zasnovani na grafovima osobina.

Pri realizaciji DNARS-a na osnovu grafa osobina korišćeni su savremeni standardi za razvoj sistema za obradu velikih količina podataka, te trenutno najbolje tehnologije. Kao rezultat, kao što je prikazano u Poglavlju 5.3, DNARS može u realnom vremenu davati odgovore na, na primer, hiljade pitanja u sekundi, pri konsultovanju baze sa preko 70 miliona rečenica. Odnosno, odgovarajuća baza je graf sa oko 60 miliona čvorova i oko 77 miliona grana; ovakav graf se, prema današnjim standardima, može okarakterisati kao *velik* (npr. McColl et al., 2014)).

## Siebog

Kao što je zaključeno u (Bordini et al., 2006; Bădică et al., 2011), trenutno postoji relativno veliki broj multiagentskih platformi – kako besplatnih, tako i onih koji se plaćaju. Međutim, nijedna postojeća platforma nije u potpunosti iskoristila savremene pristupe razvoju softvera. Iako su u postojećim sistemima uloženi neki napori kako bi se dodala podrška za veb pristup, to je najčešće učinjeno na

neefikasan način. Na primer, sistemi kao što su JADE (Bellifemine et al., 2007) i *JaCa-Web* (Minotti et al., 2010) koriste *Java aplete* za čije izvršavanje je neophodan odgovarajući dodatak (eng. *plug-in*) za veb pretraživač. Ovakvi dodaci, pak, nisu dostupni za pojedine popularne uređaje (poput *iOS*-a i "pametnih" televizora), te je i primena ovakvih sistema ograničena.

Siebog je multiagentska platforma koja pruža infrastrukturu za izvršavanja agenata u veb okruženjima, ali u skladu sa savremenim standardima i specifikacijama. Sistem uključuje dve osnovne komponente – serversku XJAF (Mitrović et al., 2012a, 2014b; Vidaković et al., 2013) i klijentsku Radigost (Mitrović et al., 2014; Mitrović et al., 2014a) – i kombinuje ih na način koji ne samo da uključuje njihove pojedinačne funkcionalnosti, već donosi i nove mogućnosti.

Na serverskoj strani, Siebog je dizajniran tako da se izvršava u klasterima računara. Na ovaj način, predloženi sistem nudi dve bitne funkcionalnosti:

- Skalabilnost: agenti se automatski raporeduju po čvorovima klastera, čime se smanjuje opterećenje pojedinačnih računara. Na taj način, Siebog je pogodan za razvoj multiagentskih sistema koji moraju da pokrenu velike brojeve agenata (npr. Panigrahi et al., 2011; Simon, 2013).

- Otpornost na greške: stanje svake serverske komponente, uključujući i agente, se kopira na preostale čvorove klastera. Ukoliko dođe do kvara na računaru, komponenta/agent može nesmetano nastaviti svoj rad na nekom od preostalih računara.

Iako podrška za distribuirano izvršavanje agenata postoji u gotovo svim multiagenskim platformama, ista je najčešće realizovana na neefikasan način. Na primer, druge platforme često implementiraju svoje pristupe za raspoređivanje agenata i otpornost na greške (npr. Alberola et al., 2013; Bellifemine et al., 2007; Faci et al., 2006; Siracuse et al., 2007). Ovaj pristup nije dovoljno fleksibilan, jer, na primer, korisnik mora sam odrediti koji agent će biti na kom računaru, dok se u slučaju Siebog-a ovaj proces odvija automatski.

Takođe, jedan od ciljeva Siebog-a je demonstracija činjenice da nije potrebno "izmišljati toplu vodu" i nanovo realizovati gorenavedene funkcionalnosti. Mnogo je efikasnije iskoristiti postojeće standarde i tehnička rešenja koja nudi *Java EE* platforma. Time se ujedno i povećava interoperabilnost Siebog-a i omogućuje se lakša integracija platforme i njenih agenata u ne-agentske sisteme koji se koriste u kompanijama.

Na serveru, Siebog je dizajniran kao modularna arhitektura. Sastoji se iz nekoliko tzv. *menadžera*, pri čemu je svaki menadžer odgovoran za određeni podskup svih funkcionalnosti. U trenutnoj realizaciji, postoje tri osnovna menadžera:

- *Menadžer agenata*: upravlja životnim ciklusima agenata i realizuje direktorijum agenata.

- *Menadžer poruka*: upravlja komunikacijom između agenata i razmenom poruka.

- *Menadžer konekcija*: služi za povezivanje distribuiranih Siebog klastera u jedinstven sistem.

Menadžeri su potpuno nezavisni jedan od drugog, a predstavljeni su i koriste se samo pomoću interfejsa, čime je postignuta maksimalna fleksibilnost u implementacijama.

Kao što je napomenuto, serverska strana Siebog-a se zasniva na ranijem sistemu XJAF, koji je prvi put predstavljen u (Vidaković and Konjović, 2002), a u međuvremenu je prošao kroz nekoliko iteracija (Mitrović et al., 2012a; Vidaković et al., 2013). U poslednjoj inkarnaciji (Mitrović et al., 2013b), koja je iskorišćena u Siebog-u, fokus je na podršci za izvršavanje u klasterima računara. Siebog, tako, podržava tzv. *simetrične klastere*, tj. klastere u kojima je svaki čvor povezan sa svakim drugim čvorom. Jedan čvor je identifikovan kao *gospodar* (eng. *master*), dok su drugi opisani kao *robovi* (eng. *slaves*). Jedina razlika između ova dva tipa čvorova je što se *gospodar* može iskoristiti za direktnu kontrolu *robova*; čvorovi su ravnopravni po svim drugim pitanjima (npr. po prioritetu izvršavanja).

Jedna od osnovnih prednosti ovakve organizacije klastera je velika fleksibilnost. Na primer, kako bi postigao otpornost na greške, Siebog ,kao što je rečeno, kopira stanja agenata na čvorove klastera. Zahvaljujući simetričnom klasteru, korisnik može birati između nekoliko načina replikacije stanja, kako bi prilagodio sistem manje, odnosno više, nestabilnim okruženjima (Surtani et al.).

Što se tiče klijentske strane, Siebog takođe nudi jedinstveni niz osobina i funkcionalnosti, poput sledećih:

- Siebog je nezavisan od platforme i podržan na velikom broju uređaja.
- Ne zahteva prethodnu instalaciju ili podešavanja. Od ovoga imaju koristi i programeri agenata, jer jednom napisanog agenta mogu koristiti na više sistema, i krajnji korisnici, jer mogu koristiti pogodnosti agentske tehnologije na najpogodniji način.
- Performanse sistema se mogu porediti sa performansama klasičnih multiagentih platformi za desktop računare (Mitrović et al., 2014a).

Trenutno postoji samo jedna multiagentska platforma sa sličnim pristupom (Jarvenpaa et al., 2013). Njen nedostatak je, međutim, što ne koristi pun skup mogućnosti koje donose *HTML5* i povezani standardi. Pored toga, nedostaju joj i ranije opisane napredne serverske mogućnosti Siebog-a.

Jedna od poteškoća u realizaciji klijentske strane Siebog-a je predstavljala činjenica da je životni vek agenta direktno vezan za veb stranicu u kojoj se izvršava. Ukoliko korisnik zatvori stranicu, agent automatski prestaje da postoji. Međutim, za bilo kakvu smisleniju primenu Siebog-a, neophodno je bilo omogućiti dugotrajno izvršavanje agenata. Ovaj problem je rešen automatskim čuvanjem internog stanja klijentskih agenata na serveru. Tačnije, kada korisnik zatvori stranicu, stanje agenta se kopira na server. Kasnije, kada korisnik ponovo učita stranicu, stanje se automatski ubacuje u agenta. Ceo proces se odvija

transparentno tj. agent nije ni "svestan" činjenice da je dolazilo do prekida u radu.

Bitno je još istaći da Siebog ne predstavlja jednostavnu uniju svojih pojedinačnih komponenti. Kao što je predstavljeno u Poglavlju 4.4, Siebog takođe nudi:

- Višeplatformsku razmenu poruka: klijentski agenti mogu nesmetano komunicirati sa serverskim agentima, pa čak i klijentskim agentima na drugim uređajima.
- Deljenje koda: jednom napisan agent se može, bez izmena, izvršavati i na serveru i na klijentu.
- Heterogena mobilnost: agenti se mogu nesmetano kretati između servera i klijenata.

Integracija klijentske i serverske strane Siebog-a je izvršena kroz nekoliko faza. U skladu sa principima integracije heterogenih sistema (Hohpe and Woolf, 2003), najpre je napravljen sloj iznad serverske strane, zasnovan na veb servisima. Tačnije, funkcionalnosti menadžera su ponuđene u obliku tzv. *RESTful* veb servisa (Fielding, 2000). Klijenska strana je proširena odgovarajućim međurealizacijama menadžera (eng. *proxy* ili *stub implementation*). Za klijentskog agenta, tako, komunikacija sa menadžerom deluje kao komunikacija sa nekom lokalnom komponentnom, a svi pozivi se u pozadini prosleđuju ka i od konkretne serverske implementacije.

Dalje, kako bi se pojednostavila višeplatformska razmena poruka, za svakog klijentskog agenta se na serveru kreira odgovarajući među-objekat. Sve poruke upućene ovom među-objektu na serveru se, ponovo u pozadini, prosleđuju odgovarajućem klijentskom agentu. Siebog tako nudi interesantne modele za komunikaciju fizički udaljenih agenata. Na primer:

- Klijentski agent šalje poruku preko Menadžera poruka sa kojim komunicira kao sa lokalnom komponentom.
- Poruka se prebacuje na server i upućuje serverskom agentu.
- Serverski agent, koji zapravo predstavlja među-objekat, prosleđuje poruku svom klijentskom agentu na udaljenom uređaju.

Na ovaj način, agenti koji su zapravo fizički udaljeni, komuniciraju kao da se nalaze na istom uređaju.

Konačno, Siebog je proširen tako da uključuje podršku za rasuđivanje zasnovano na DNARS-u. Na serverskoj strani je primenjena čvršća integracija dva sistema, kako bi se smanjili troškovi njihove komunikacije, te da bi mogle biti ponuđene programske apstrakcije za definisanje agenata. Pošto su *anotacije* jedan od standardnih tehnika za meta-programiranje u *Java EE* platformi, isti pristup je primenjen i prilikom integracije serverskog Siebog-a i DNARS-a; tj. programerima je ponuđen niz anotacija za označavanje npr. ciljeva

agenta. Sistem zatim u pozadini automatski prevodi signale koje šalje DNARS-ov Rukovodilac događajima u pozive odgovarajućih metoda.

Integracija klijentske strane Siebog-a sa DNARS-om je realizovana na sličan način kao i integracija sa serverskom stranom Siebog-a, odnosno kroz komunikaciju sa odgovarajućim veb servisima.

Po pitanju podrške za inteligentne agente, Siebog, dakle, "napušta" tradicionalnu BDI arhitekturu. Umesto toga, ponuđena je mogućnost razvoja inteligentnih multiagentskih sistema koji uključuju agente sa inovativnim sposobnostima rasuđivanja. Jedan konkretan primer praktične primene ovakvog pristupa, dat u Glavi 5, bi, na primer, teško mogao biti rešen nekim od postojećih sistema.

Na ovaj način, Siebog i DNARS predstavljaju jedinstven multiagenski okvir za razvoj i izvršavanje inteligentnih agenata u veb okruženjima, te nude nove i zanimljive načine za praktične primene agentske tehnologije. Primer generisanja novog struktuiranog znanja za projekat *DBpedia*-e (Lehmann et al., 2014) je opisan u Pogljavlju 5.4. Dodatne mogućnosti primena uključuju, na primer, razvoj personalnih inteligentnih asistenata (overview of agents in knowledge management, 2006), sisteme sa velikim agenatskim društvima (Ilie et al., 2011), razvoj inteligentnih komponenti u domenima *Interneta Stvari* (eng. *Internet of Things*) i tzv. "pametnim" okruženjima (Nakashima et al., 2010), itd.

# Bibliography


J. M. Alberola, J. M. Such, V. Botti, A. Espinosa, and A. Garcia-Fornes. A scalable multiagent platform for large systems. *Computer Science and Information Systems*, 10(1):51–77, January 2013. doi: 10.2298/CSIS111029039A.

E. Alpaydin. *Introduction to machine learning*. The MIT Press, 2004.

J. R. Anderson, D. Bothell, and M. D. Byrne. An integrated theory of mind. *Psychological review*, 111(4):1036–1060, 2004.

G. Aranda, J. Palanca, M. Escriva, and J. A. Garcia-Pardo. *SPADE user's manual*, 2012. URL `https://pythonhosted.org/SPADE/index.html`. Retrieved on February 7, 2014.

E. Argente, J. Palanca, G. Aranda, V. Julian, V. Botti, A. Garcia-Fornes, and A. Espinosa. Supporting agent organizations. In H.-D. Burkhard, G. Lindemann, R. Verbrugge, and L. Varga, editors, *Multi-Agent Systems and Applications V*, volume 4696 of *Lecture Notes in Computer Science*, pages 236–245. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75253-0. doi: 10.1007/978-3-540-75254-7_24.

B. Baesens. *Analytics in a Big Data World: The Essential Guide to Data Science and its Applications*. Wiley and SAS Business Series, 2014.

*Cougaar Developers' Guide*. BBN Technologies, December 2004. URL `http://cougaar.org/doc/11_4/online/CDG_11_4.pdf`. Retrieved on February 7, 2014.

F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. Jade – a java agent development framework. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer US, 2005.

F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

C. Blum and D. Merkle, editors. *Swarm intelligence: introduction and applications*. Springer-Verlag Berling Heidelberg, 2008.

O. Boissier, R. H. Bordini, J. F. Hubner, A. Ricci, and A. Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78 (6):747–761, 2013. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2011.10.004.

R. H. Bordini and J. F. Hubner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-33996-0.

R. H. Bordini, L. Braubach, M. Dastani, A. El, F. Seghrouchni, J. J. Gomez-sanz, J. Leite, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

R. H. Bordini, J. F. Hubner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. John Wiley & Sons Ltd, 2007. ISBN 978-0-470-02900-8.

R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors. *Multi-agent programming: languages, tools and applications*. Springer, 2009. ISBN 978-0-387-89298-6. doi: 10.1007/978-0-387-89299-3.

M. Born, editor. *The Born-Einstein Letters*. Walker, 1971.

L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex active components: A unified execution infrastructure for agents and workflows. *Advanced Computational Technologies*, pages 128–149, 2013.

D. Brickley and R. Guha. RDF Schema 1.1. `http://www.w3.org/TR/rdf-schema/`.

T. L. Briggs and J. Werth. A specification language for object-oriented analysis and design. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP '94, pages 365–385, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-58202-9.

R. A. Brooks. Intelligence without representation. *Artif. Intell.*, 47(1-3):139–159, February 1991. ISSN 0004-3702. doi: 10.1016/0004-3702(91)90053-M.

C. Bădică, Z. Budimac, H.-D. Burkhard, and M. Ivanović. Software agents: languages, tools, platforms. *Computer Science and Information Systems, ComSIS*, 8(2):255–298, 2011.

L. Busoniu, R. Babuska, and B. De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(2):156–172, March 2008. ISSN 1094-6977. doi: 10.1109/TSMCC.2007.913919.

G. Cabri, L. Leonardi, and F. Zambonelli. Weak and strong mobility in mobile agent applications. In *Proceedings of the 2nd international conference and exhibition on the practical applications of Java*, 2000.

A. Carzaniga, G. P. Picco, and G. Vigna. Is code still moving around? looking back at a decade of code mobility. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 9–20, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2892-9. doi: 10.1109/ICSECOMPANION.2007.44.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2): 4:1–4:26, 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816.

H.-Q. Chong, A.-H. Tan, and G.-W. Ng. Integrated cognitive architectures: a survey. *Artificial Intelligence Review*, 28:103–130, 2007.

P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0.

D. D. Corkill. Collaborative software: blackboard and multi-agent systems & the future. In *Proceedings of the International Lisp Conference*, October 2003.

M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. C. Meyer. A programming language for cognitive agents - goal directed 3APL. In *PROMAS*, pages 111–130, 2003.

W. H. E. Davies and P. Edwards. Agent-K: An integration of AOP and KQML. In *Proceedings of the Third International Conference on Information and Knowledge Management*, 1994.

M. de Weerdt and B. Clement. Introduction to planning in multiagent systems. *Multiagent Grid Syst.*, 5(4):345–355, 2009. ISSN 1574-1702.

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492.

L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans, version 3.0. `http://download.oracle.com/otndocs/jcp/ejb-3_0-fr-eval-oth-JSpec/`, May 2006.

M. D'Inverno, D. Kinny, and M. Luck. Interaction protocols in Agentis. In *Proceedings of the Third International Conference on Multi-Agent Systems*, pages 112–119. IEEE Press, 1998.

M. D'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge. The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004. ISSN 1387-2532. doi: 10.1023/B:AGNT.0000019688.11109.19.

M. Dorigo and T. St uzle. *Ant colony optimization*. The MIT Press, 2004.

W. Duch, R. J. Oentaryo, and M. Pasquier. Cognitive architectures: Where do we go from here? In *Proceedings of the 2008 Conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, pages 122–136, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press. ISBN 978-1-58603-833-5.

B. M. Dunin-Keplicz and R. Verbrugge. *Teamwork in Multi-Agent Systems: A Formal Approach*. Wiley Publishing, 1st edition, 2010. ISBN 0470699884, 9780470699881.

N. Faci, Z. Guessoum, and O. Marin. DimaX: A fault-tolerant multi-agent platform. In *Proceedings of the 2006 International Workshop on Software Engineering for Large-scale Multi-agent Systems*, SELMAS '06, pages 13–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-395-6.

R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

FIPA Acl. FIPA ACL message structure specification. `http://www.fipa.org/specs/fipa00061/SC00061G.pdf`, 2002. Retrieved on February 7, 2014.

FIPA Act. FIPA communicative act library specification. `http://www.fipa.org/specs/fipa00037/SC00037J.pdf`, 2002. Retrieved on February 7, 2014.

FIPA Ams. FIPA agent management specification. `http://www.fipa.org/specs/fipa00023/SC00023K.pdf`, 2004. Retrieved on February 7, 2014.

FIPA CNet. FIPA Contract Net interaction protocol specification. `http://www.fipa.org/specs/fipa00029/SC00029H.pdf`. Retrieved on February 7, 2014.

FIPA Home. FIPA homepage. `http://www.fipa.org/`. Retrieved on February 7, 2014.

FIPA Mts. FIPA agent message transport service specification. `http://www.fipa.org/specs/fipa00067/SC00067F.pdf`, 2002. Retrieved on February 7, 2014.

G. Fortino, A. Garro, and W. Russo. Achieving mobile agent systems interoperability through software layering. *Information and Software Technology*, 50 (4):322–341, 2008. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2007.02.016.

S. Franklin. A foundational architecture for artificial general intelligence. In *Proceedings of the 2007 Conference on Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms: Proceedings of the AGI Workshop 2006*, pages 36–54, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press. ISBN 978-1-58603-758-1.

T. Gabel and M. Riedmiller. On a successful application of multi-agent reinforcement learning to operations research benchmarks. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 68–75, 2007.

M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, 1987.

C. Georgousopoulos, O. F. Rana, and A. Karageorgos. Supporting fipa interoperability for legacy multi-agent systems. In P. Giorgini, J. P. M uller, and J. Odell, editors, *Agent-Oriented Software Engineering IV*, volume 2935 of *Lecture Notes in Computer Science*, pages 167–184. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-20826-6. doi: 10.1007/978-3-540-24620-6_12.

T. Gherbi, I. Borne, and D. Meslati. MDE and mobile agents: another reflection on the agent migration. In *Proceedings of the 11th international conference on computer modelling and simulation*, pages 468–473, 2009.

S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700. doi: 10.1145/564585.564601.

B. Goertzel. OpenCog Prime: A cognitive synergy based architecture for embodied artificial general intelligence. In *Proceedings of ICCI-09*, 2009.

B. Goertzel, H. de Garis, C. Pennachin, N. Geisweiller, S. Araujo, J. Pitt, S. Chen, R. Lian, M. Jiang, Y. Yang, and D. Huang. OpenCogBot: Achieving generally intelligent virtual agent control and humanoid robotics via cognitive synergy. In *Proceedings of ICAI-10*, 2010a.

B. Goertzel, R. Lian, I. Arel, H. de Garis, and S. Chen. A world survey of artificial brain projects, part ii: Biologically inspired cognitive architectures. *Neurocomputing*, 74(1–3):30–49, 2010b.

A. Goncalves. *Beginning Java EE 6 platform with GlassFish 3, second edition*. apress, 2010.

A. Grimstrup, R. S. Gray, D. Kotz, M. M. Carvalho, T. B. Cowin, D. A. Chacón, J. Barton, C. Garrett, and M. Hofmann. Toward interoperability of mobile-agent systems. In *International symposium on mobile agents*, volume 2002, pages 106–120.

M. Grogan. *JSR-223: Scripting for the Java platform.* Sun Microsystems, Inc., 2006. URL `https://jcp.org/aboutJava/communityprocess/final/jsr223/index.html`. Retrieved on January 22, 2014.

S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

A. Guerra-Hernandez, J. M. Castro-Manzano, and A. E. F. Seghrouchni. CTL AgentSpeak(L): A specification language for agent programs. *Journal of Algorithms Cognition, Informatics and Logic*, 60:31–40, 2009.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983. ISSN 0360-0300. doi: 10.1145/289.291.

D. Hart and B. Goertzel. OpenCog: A software framework for integrative artificial general intelligence. In *Proceedings of the 2008 Conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, pages 468–472, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press. ISBN 978-1-58603-833-5.

B. Henderson-Sellers and P. Giorgini, editors. *Agent-oriented methodologies.* Idea Group Publishing, 2005.

C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on artificial intelligence (IJCAI'73)*, pages 235–245, 1973.

E. Hewitt. *Cassandra: the definitive guide.* O'Reilly Media, Inc., 2010.

K. Hindriks. Programming cognitive agents in GOAL. `http://mmi.tudelft.nl/trac/goal/raw-attachment/wiki/WikiStart/Guide.pdf`, March 2014. Retrieved on February 7, 2014.

K. V. Hindriks. Programming rational agents in GOAL. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009. ISBN 978-0-387-89298-6. doi: 10.1007/978-0-387-89299-3_4.

G. Hohpe and B. Woolf. *Enterprise integration patterns: designing, building, and deploying messaging solutions.* Addison Wesley, 2003.

E. Hovy, R. Navigli, and S. P. Ponzetto. Collaboratively built semi-structured content and artificial intelligence: The story so far. *Artificial Intelligence*, 194: 2–27, 2013. ISSN 0004-3702.

M. N. Huhns and L. M. Stephens. Multiagent systems and societies of agents. In G. Weiss, editor, *Multiagent systems: a modern approach to distributed artificial intelligence*, chapter 2, pages 79–120. The MIT Press, 1999. ISBN 0-262-23203-0.

S. Ilie and C. Badica. A comparison of the island and ACODA approaches for distributing ACO. In *17th International Conference System Theory, Control and Computing (ICSTCC)*, pages 757–762, October 2013.

S. Ilie, A. Bădică, and C. Bădică. Distributed agent-based ant colony optimization for solving traveling salesman problem on a partitioned map. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, WIMS '11, pages 23:1–23:9, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0148-0. doi: 10.1145/1988688.1988716.

Jack WebBot. JACK Intelligent Agents WebBot manual. `http://www.aosgrp.com/documentation/jack/WebBot_Manual_WEB/index.html`, March 2009. Retrieved on February 7, 2014.

L. Jarvenpaa, M. Lintinen, A.-L. Mattila, T. Mikkonen, K. Systa, and J.-P. Voutilainen. Mobile agents for the internet of things. In *17th International Conference on System Theory, Control and Computing*, pages 763–767. 2013.

D. Jarvis, J. Jarvis, R. Ronnquist, and L. Jain. *Multiagent system and applications: development using the GORITE BDI Framework*, volume 46 of *Intelligent System Reference Library*. Springer, 2010.

G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

V. Kelemen. JADE tutorial: simple example for using the JadeGateway class. `http://jade.cselt.it/doc/tutorials/JadeGateway.pdf`, October 2006. Retrieved on February 7, 2014.

J. Kennedy and R. C. Eberhart. Particle swarm optimizatio. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

D. Kinny. The Agentis agent interaction model. In J. P. Muller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V. Agents Theories, Architectures, and Languages*, pages 331–344. Springer-Verlag Berlin Heidelberg, 1999.

J. Kodama, T. Hamagami, H. Shinji, T. Tanabe, T. Funabashi, and H. Hirata. Multi-agent-based autonomous power distribution network restoration using contract net protocol. *Electrical Engineering in Japan*, 166(4):56–63, 2009.

J. Kuhlenkamp, M. Klems, and O. Röss. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment*, 7(12): 1219–1230, 2014. ISSN 2150-8097.

M. Kumar. *Quantum: Einstein, Bohr and the great debate about the nature of reality*. Icon Books Ltd., 2009.

J. E. Laird. Extending the Soar cognitive architecture. In *Proceedings of the First Conference on Artificial General Intelligence*, pages 224–235, 2008.

J. E. Laird. *The Soar cognitive architecture*. MIT Press, 2012.

J. E. Laird, K. R. Kinkade, S. Mohan, and J. Z. Xu. Cognitive robotics using the Soar cognitive architecture. In *8th International Conference on Cognitive Robotics (Cognitive Robotics Workshop, 26th Conference on Artificial Intelligence)*, 2012.

P. Langley and D. Choi. A unified cognitive architecture for physical agents. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, AAAI'06, pages 1469–1474. AAAI Press, 2006. ISBN 978-1-57735-281-5.

C. Lebiere and J. R. Anderson. A connectionist implementation of the ACT-R production system. In *Proceedings of the 15th Annual Conference of the Cognitive Science Society*, pages 635–640, 1993.

J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 2014.

M. Luck, R. Ashri, and M. d'Inverno. *Agent-based software development*. Agent-Oriented Systems. Artech House Publishers, 2004.

C. M. Macal and M. J. North. Agent-based modeling and simulation. In *Winter Simulation Conference*, WSC '09, pages 86–98. Winter Simulation Conference, 2009. ISBN 978-1-4244-5771-7.

P. Maes. The agent network architecture (ana). *SIGART Bull.*, 2(4):115–120, July 1991. ISSN 0163-5719. doi: 10.1145/122344.122367.

G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184.

F. Marchioni. *WildFly 8 Administration*. ItBuzzPress, 2014.

F. Marchioni and M. Surtani. *Infinispan data grid platform*. Packt Publishing Ltd., 2012.

V. Mascardi, D. Demergasso, and D. Ancona. Languages for programming BDI-style agents: an overview. In *WOA 2005*, pages 9–15, 2005.

J. McCarthy. What is artificial intelligence? `http://www-formal.stanford.edu/jmc/whatisai/whatisai.html`, November 2007. Retrieved on February 10, 2014.

R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 11–18, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2654-4. doi: 10.1145/2567634.2567638.

P. McCorduck. *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. AK Peters Ltd, 2004. ISBN 1568812051.

N. Medvidovic and G. Edwards. Software architecture and mobility: A roadmap. *Journal of Systems and Software*, 83(6):885–898, June 2010. ISSN 0164-1212. doi: 10.1016/j.jss.2009.11.004.

P. N. Mendes, M. Jakob, A. García-Silva, and C. Bizer. Dbpedia spotlight: Shedding light on the web of documents. In *Proceedings of the 7th International Conference on Semantic Systems (I-Semantics)*, 2011.

P. N. Mendes, M. Jakob, and C. Bizer. DBpedia for NLP: A multilingual cross-domain knowledge base. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, 2012. ISBN 978-2-9517408-7-7.

F. R. Meneguzzi, A. F. Zorzo, M. da Costa Mora, and M. Luck. Incorporating planning into BDI systems. *Scalable computing: practice and experience*, 8(1): 15–28, 2007.

M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using Nutch/Lucene. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

M. Minotti, A. Santi, and A. Ricci. Developing web client applications with JaCa-Web. In A. Omicini and M. Viroli, editors, *Proceedings of the 11th WOA 2010 Workshop, Dagli Oggetti Agli Agenti, Rimini, Italy, September 5-7, 2010*, volume 621 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

D. Mitrović, M. Ivanović, Z. Budimac, and M. Vidaković. An overview of agent mobility in heterogeneous environments. In *Proceedings of the Workshop on Applications of Software Agents (WASA 2011)*, pages 52–58, 2011.

D. Mitrović, M. Ivanović, and M. Vidaković. Introducing ALAS: a novel agent-oriented programming language. In T. E. Simos, editor, *Proceedings of Symposium on Computer Languages, Implementations, and Tools (SCLIT 2011)*, volume 1389 of *AIP Conference Proceedings*, pages 861–864, September 2011.

D. Mitrović, M. Ivanović, Z. Budimac, and M. Vidaković. Supporting heterogeneous agent mobility with ALAS. *Computer Science and Information Systems*, 9(3):1203–1229, 2012a.

D. Mitrović, M. Ivanović, M. Vidaković, and A. Al-Dahoud. Developing software agents using Enterprise JavaBeans. In *Local Proceedings of the Fifth Balkan Conference in Informatics (BCI 2012)*, pages 147–149, 2012b.

D. Mitrović, M. Ivanović, and C. Bădică. Jason agents in Java EE environments. In E. Petre and M. Brezovan, editors, *3rd Workshop on Applications of Software Agents (WASA 2013), held within 17th International Conference on System Theory, Control and Computing (ICSTCC 2013)*, pages 768–771, Sinaia, Romania, October 2013a.

D. Mitrović, M. Ivanović, and H.-D. Burkhard. Intelligent Jason agents in virtual soccer simulations. In M. Klusch, M. Thimm, and M. Paprzycki, editors, *11th German Conference on Multiagent System Technologies*, volume 8076 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 2013b.

D. Mitrović, M. Ivanović, and C. Bădică. Delivering the multiagent technology to end-users through the web. In *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14)*, WIMS '14, pages 54:1–54:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2538-7. doi: 10.1145/2611040.2611102.

D. Mitrović, M. Ivanović, Z. Budimac, and M. Vidaković. Radigost: Interoperable web-based multi-agent platform. *Journal of Systems and Software*, 90:167–178, 2014a. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2013.12.029.

D. Mitrović, M. Ivanović, M. Vidaković, and Z. Budimac. Extensible Java EE-based agent framework in clustered environments. In J. Mueller, M. Weyrich, and A. L. C. Bazzan, editors, *12th German Conference on Multiagent System Technologies*, volume 8732 of *Lecture Notes in Computer Science*, pages 202–215. Springer International Publishing, 2014b.

M. C. Mora, J. G. Lopes, R. M. Viccariz, and H. Coelho. BDI models and systems: Reducing the gap. In J. P. M uller, A. S. Rao, and M. P. Singh, editors, *Intelligent Agents V: Agents Theories, Architectures, and Languages*, volume 1555 of *Lecture Notes in Computer Science*, pages 11–27. Springer Berlin Heidelberg, 1999.

G. Mulligan and D. Gracanin. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In *Proceedings of the 2009 Winter Simulation Conference*, pages 1423–1432. IEEE, 2009.

H. Nakashima and I. Noda. Dynamic subsumption architecture for programming intelligent agents. In *Proceedings of the International Conference on Multi Agent Systems*, pages 190–197, 1998. doi: 10.1109/ICMAS.1998.699049.

H. Nakashima, H. Aghajan, and J. C. Augusto. *Handbook of ambient intelligence and smart environments*. Springer, 2010.

P. Nathan. *Enterprise data workflows with Cascading*. O'Reilly Media, Inc., 2013.

A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1994.

O. Nichifor and S. Buraga. ADF – abstract framework for developing mobile agents. Technical Report TR 04-01, Faculty of Computer Science, "A. I. Cuza" University of Iasi, Romania, 2004.

G. S. Nitschke, A. E. Eiben, and M. C. Schut. Evolving team behaviors with specialization. *Genetic Programming and Evolvable Machines*, 13(4):493–536, December 2012. ISSN 1389-2576. doi: 10.1007/s10710-012-9166-5.

S. Nolfi. Power and the limits of reactive agents. *Neurocomputing*, 42(1):119–145, 2002. ISSN 0925-2312. doi: http://dx.doi.org/10.1016/S0925-2312(01)00598-7. Evolutionary neural systems.

I. Nunes, C. J. de Lucena, and M. Luck. BDI4JADE: a BDI layer on top of JADE. In *Proceedings of the Workshop on Programming Multiagent Systems*, pages 88–103, 2011.

R. Oentaryo and M. Pasquier. Towards a novel integrated neuro-cognitive architecture (INCA). In *IEEE International Joint Conference on Neural Networks*, pages 1902–1909, June 2008.

S. Okamoto and M. Kohana. Load distribution by using web workers for a real-time web application. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pages 592–597, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0421-4. doi: http://doi.acm.org/10.1145/1967486.1967577.

B. J. Overeinder, D. R. A. D. Groot, N. J. E. Wijngaards, and F. M. T. Brazier. Generative mobile agent migration in heterogeneous environments. *Scalable computing: practice and experience*, 7(4):89–99, 2006.

A. overview of agents in knowledge management. V. dignum. In *Proceedings of INAP-05*, volume 4369 of *Lecture Notes in Artificial Intelligence*, pages 175–189, 2006.

L. Panait and S. Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, November 2005. doi: 10.1007/s10458-005-2631-2.

B. K. Panigrahi, Y. Shi, , and M.-H. Lim, editors. *Handbook of swarm intelligence: concepts, principles and applications*. Springer-Verlag Berling Heidelberg, 2011.

H. Paulheim and C. Bizer. Improving the quality of linked data using statistical distributions. *International Journal of Semantic Web and Information Systems*, 10(2):63–86, 2014.

D. Pereira, E. Oliveira, and N. Moreira. Formal modelling of emotions in bdi agents. In F. Sadri and K. Satoh, editors, *Computational Logic in Multi-Agent Systems*, volume 5056 of *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88832-1. doi: 10.1007/978-3-540-88833-8_4.

T. Phung, M. Winikoff, and L. Padgham. Learning within the bdi framework: An empirical analysis. In R. Khosla, R. Howlett, and L. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 3683 of *Lecture Notes in Computer Science*, pages 282–288. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28896-1. doi: 10.1007/11553939_41.

U. Pinsdorf and V. Roth. Mobile agent interoperability patterns and practice. In *Proceedings of the 9th IEEE international conference on engineering of computer-based systems*, pages 238–244, 2002.

A. Pokahr and L. Braubach. From a research to an industry-strength agent platform: Jadex V2. In *9. Internationale Tagung Wirtschaftsinformatik*, pages 769–778, 2008.

A. Pokahr, L. Braubach, C. Haubeck, and J. Ladiges. Programming bdi agents with pure java. In J. P. Müller, M. Weyrich, and A. L. Bazzan, editors, *Multiagent System Technologies*, volume 8732 of *Lecture Notes in Computer Science*, pages 216–233. Springer International Publishing, 2014. ISBN 978-3-319-11583-2. doi: 10.1007/978-3-319-11584-9_15.

T. A. Polk and C. M. Seifert, editors. *Cognitive modeling*. MIT Press, 2002.

M. E. Pollack, D. Joslin, A. Nunes, S. Ur, and E. Ephrati. Experimental investigation of an agent-commitment strategy. Technical Report 94–31, University of Pittsburgh, Dept. of Computer Science, 1994.

J. Preston and M. Bishop, editors. *Views into the Chinese Room: New Essays on Searle and Artificial Intelligence*. Oxford University Press, 2002.

D. Purdy and J. Richter. Exploring the Observer design pattern. `https://msdn.microsoft.com/en-us/library/Ee817669(pandp.10).aspx`. Retrieved on April 22, 2014.

T. Rabl, S. Gomez-Villamor, M. Sadoghi, V. Muntes-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367512.

A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW*, pages 42–55, 1996.

A. S. Rao and M. P. Georgeff. Intentions and rational commitment. Technical Report 8, Australian Artificial Intelligence Institute, 1993.

A. S. Rao and M. P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, 1995.

G. Rimassa, M. Calisti, and M. E. Kernland. Living systems ®technology suite. In R. Unland, M. Klusch, and M. Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 73–93. Birkhauser Verlag, 2005.

I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. O'Reilly Media, Inc., 2013.

M. A. Rodriguez and J. Geldart. An evidential path logic for multi-relational networks. In *Proceedings of the Association for the Advancement of Artificial Intelligence Spring Symposium: Technosocial Predictive Analytics Symposium*, volume SS-09-09, pages 114–119. AAAI Press, 2009.

M. A. Rodriguez and J. Shinavier. Exposing multi-relational networks to single-relational network analysis algorithms. *Journal of Informetrics*, 4(1):29–41, 2010. ISSN 1751-1577. doi: http://dx.doi.org/10.1016/j.joi.2009.06.004.

J. L. Rodriguez-Fernandez. Ockham's razor. *Endeavour*, 23(3):121–125, 1999. ISSN 0160-9327. doi: http://dx.doi.org/10.1016/S0160-9327(99)01199-0.

R. R onnquist. The goal oriented teams (gorite) framework. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Programming Multi-Agent Systems*, volume 4908 of *Lecture Notes in Computer Science*, pages 27–41. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-79042-6. doi: 10.1007/978-3-540-79043-3_2.

P. J. Sadalage and M. Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 2012.

I. Sakellariou. Controlling turtles through state machines: An application to pedestrian simulation. In M. S. Obaidat, J. Filipe, J. Kacprzyk, and N. Pina, editors, *Simulation and Modeling Methodologies, Technologies and Applications*, volume 256 of *Advances in Intelligent Systems and Computing*, pages 197–210. Springer International Publishing, 2014. doi: 10.1007/978-3-319-03581-9_14.

T. Salamon. *Design of Agent-Based Models: Developing Computer Simulations for a Better Understanding of Social Processes*. Academic series. Bruckner Publishing, Repin, Czech Republic, 9 2011. ISBN 978-80-904661-1-1.

A. Schram and K. M. Anderson. Mysql to nosql: Data modeling challenges in supporting scalability. In *Proceedings of the 3rd Annual Conference on*

*Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 191–202, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1563-0. doi: 10.1145/2384716.2384773.

G. Schreiber and Y. Raimond. RDF 1.1 primer. `http://www.w3.org/TR/rdf11-primer/`.

S. Sen and G. Weiss. Learning in multiagent systems. In G. Weiss, editor, *Multiagent systems: a modern approach to distributed artificial intelligence*, chapter 6, pages 259–298. The MIT Press, 1999. ISBN 0-262-23203-0.

A. Shalyto, L. Naumov, and G. Korneev. Methods of object-oriented reactive agents implementation on the basis of finite automata. In *Integration of Knowledge Intensive Multi-Agent Systems, 2005. International Conference on*, pages 460–465, April 2005. doi: 10.1109/KIMAS.2005.1427125.

N. E. Sharkey and T. Ziemke. Mechanistic versus phenomenal embodiment: Can robot embodiment lead to strong ai? *Cognitive Systems Research*, 2(4):251–262, 2001. ISSN 1389-0417. doi: http://dx.doi.org/10.1016/S1389-0417(01)00036-5.

Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2008.

D. Simon. *Evolutionary optimization algorigthms*. Wiley, 2013.

G. Simon. Mastering Big Data: CFO strategies to transform insight into opportunity. Technical report, FSN & Oracle White Paper, 2012.

D. Singh, S. Sardina, L. Padgham, and G. James. Integrating learning into a bdi agent for environments with changing dynamics. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pages 2525–2530. AAAI Press, 2011. ISBN 978-1-57735-515-1. doi: 10.5591/978-1-57735-516-8/IJCAI11-420.

M. P. Singh, A. S. Rao, and M. P. Georgeff. Formal methods in DAI: Logic-based representation and reasoning. In G. Weiss, editor, *Multiagent systems: a modern approach to distributed artificial intelligence*, chapter 8, pages 331–376. The MIT Press, 1999. ISBN 0-262-23203-0.

S. Siracuse, R. Tomlinson, T. Wright, and J. Zinky. Experience with task/allocation coordination primitive for building survivable multi-agent systems. In *International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, pages 40–45, April 2007.

R. Smith. Aristotle's logic. The Stanford Encyclopedia of Philosophy, 2012. URL `http://plato.stanford.edu/cgi-bin/encyclopedia/archinfo.cgi?entry=aristotle-logic`. Retrieved on January 22, 2014.

F. Sommers and G. Englebretsen. *An invitation to formal reasoning: the logic of terms*. Ashgate Publishing Ltd, 2000.

N. Suri, J. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, and R. Jeffers. Strong mobility and fine-grained resource control in NOMADS. In D. Kotz and F. Mattern, editors, *Proceedings of the Second international Symposium on Agent Systems and Applications and Fourth international Symposium on Mobile Agents*, volume 1882 of *Lecture Notes In Computer Science*, pages 2–15, September 2000.

M. Surtani, M. Markus, G. Zamarreno, and P. Muir. Infinispan user guide. `http://infinispan.org/docs/6.0.x/user_guide/user_guide.html`. Retrieved on April 22, 2014.

A. Tacy, R. Hanson, J. Essington, and A. T okke. *GWT in action*. Manning Publications, second edition edition, 2014.

J. Tardo and L. Valente. Mobile agent security and telescript. In *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pages 58–63, February 1996. doi: 10.1109/CMPCON.1996.501749.

S. R. Thomas. The PLACA agent programming language. In *ECAI Workshop on Agent Theories, Architectures, and Languages*, pages 355–370, 1994.

K. R. Thorisson and H. P. Helgasson. Cognitive architectures and autonomy: A comparative review. *Journal of Artificial General Intelligence*, 3(2):1–30, 2012.

S. Tiwari. *Professional NoSQL*. John Wiley & Sons, Inc., 2011.

A. M. Turing. Computing machinery and intelligence. *Mind*, LIX:433–460, 1950.

O. Urra, S. Ilarri, T. Delot, and E. Mena. Mobile agents in vehicular networks: Taking a first ride. In Y. Demazeau, F. Dignum, J. Corchado, and J. Perez, editors, *Advances in Practical Applications of Agents and Multiagent Systems*, volume 70 of *Advances in Intelligent and Soft Computing*, pages 119–124. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12383-2. doi: 10.1007/978-3-642-12384-9_15.

L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33 (8):103–111, 1990. ISSN 0001-0782. doi: 10.1145/79173.79181.

W. van der Hoek and M. Wooldridge. Logics for multiagent systems. *AI Magazine*, 33(3):92–105, 2012.

W. van der Hoek and M. Wooldridge. Logics for multiagent systems. In G. Weiss, editor, *Multiagent Systems*, Intelligent robotics and autonomous agents, chapter 16, pages 761–811. The MIT Press, 2013.

M. Vidaković and Z. Konjović. EJB based intelligent agents framework. In *The 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 343–348, 2002.

M. Vidaković, M. Ivanović, D. Mitrović, and Z. Budimac. Extensible Java EE-based agent framework – past, present, future. In M. Ganzha and L. C. Jain, editors, *Multiagent Systems and Applications*, volume 45 of *Intelligent Systems Reference Library*, pages 55–88. Springer Berlin Heidelberg, 2013.

N. Vlassis. *A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning (Book 2). Morgan and Claypool Publishers, 2007.

J. Vokřínek, J. Bíba, J. Hodík, J. Vybíhal, and M. Pěchouček. Competitive contract net protocol. In *SOFSEM 2007: Theory and Practice of Computer Science*, pages 656–668. Springer, 2007.

P. Wang. From inheritance relation to non-axiomatic logic. *International Journal of Approximate Reasoning*, 11(4):281–319, 1994.

P. Wang. Unified inference in extended syllogism. In P. Flach and A. Kakas, editors, *Abduction and Induction*, volume 18 of *Applied Logic Series*, pages 117–129. Springer Netherlands, 2000. ISBN 978-90-481-5433-3. doi: 10.1007/978-94-017-0606-3_8.

P. Wang. Abduction in non-axiomatic reasoning. In *Working Notes of the IJCAI workshop on Abductive Reasoning*, pages 56–63, 2001a.

P. Wang. Confidence as higher order uncertainty. In *Proceedings of the Second International Symposium on Imprecise Probabilities and Their Applications*, pages 352–361, 2001b.

P. Wang. Experience-grounded semantics: a theory for intelligent systems. *Cognitive Systems Research*, 6(4):282–302, 2005. ISSN 1389-0417. doi: http://dx.doi.org/10.1016/j.cogsys.2004.08.003.

P. Wang. *Rigid flexibility: the logic of intelligence*, volume 34 of *Applied Logic Series*. Springer, Dordrecht, The Netherlands, 2006. ISBN 978-1-4020-5045-9.

P. Wang. The logic of intelligence. In *In Ben Goertzel and Cassio Pennachin, editors, Artificial General Intelligence*, pages 31–62. Springer, 2007.

P. Wang. Analogy in a general-purpose reasoning system. *Cognitive Systems Research*, 10(3):286–296, September 2009. ISSN 1389-0417.

P. Wang. Formalization of evidence: a comparative study. *Journal of Artificial General Intelligence*, 1(1):25–53, November 2011. ISSN 1946-0163. doi: 10.2478/v10229-011-0003-7.

P. Wang. Theories of artificial intelligence – meta-theoretical considerations. In P. Wang and B. Goertzel, editors, *Theoretical Foundations of Artificial General Intelligence*, volume 4 of *Atlantis Thinking Machines*, pages 305–323. Atlantis Press, 2012a. ISBN 978-94-91216-61-9. doi: 10.2991/978-94-91216-62-6_16.

P. Wang. Solving a problem with or without a program. *Journal of Artificial General Intelligence*, 3(3):43–73, 2012b.

P. Wang. *Non-axiomatic logic: a model of intelligent reasoning*. World Scientific Publishing Co. Pte. Ltd., 2013.

P. Wang and S. Awan. Reasoning in non-axiomatic logic: A case study in medical diagnosis. In J. Schmidhuber, K. R. Thorisson, and M. Looks, editors, *Artificial General Intelligence*, volume 6830 of *Lecture Notes in Computer Science*, pages 297–302. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22886-5. doi: 10.1007/978-3-642-22887-2_33.

G. Weiss, editor. *Multiagent systems: a modern approach to distributed artificial intelligence*. The MIT Press, 1999. ISBN 0-262-23203-0.

G. Weiss, editor. *Multiagent systems*. Intelligent robotics and autonomous agents. The MIT Press, 2013. ISBN 978-0-262-01889-0.

T. White. *Hadoop: the definite guide*. O'Reilly Media, Inc., 3rd edition, 2012.

M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent systems: a modern approach to distributed artificial intelligence*, chapter 1, pages 27–78. The MIT Press, 1999. ISBN 0-262-23203-0.

M. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 213–220, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1851-8.

X. Yan, C. Zhang, W. Luo, W. Li, W. Chen, and H. Liu. Solve traveling salesman problem using particle swarm optimization algorithm. *International Journal of Computer Science Issues*, 9(6):264–271, 2012.

D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 237–249, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190252.

## Kratka biografija

Dejan Mitrović je rođen 28. decembra 1982. godine u Zrenjaninu. Na Univerzitetu u Novom Sadu je 2001. godine upisao Prirodno-matematički fakultet, smer Diplomirani informatičar – poslovna informatika. Studije je završio 2006. godine odbranivši diplomski rad pod nazivom "Efikasno renderovanje velikih terena". Master studije informatike, smer Diplomirani informatičar – master, je završio 2008. godine odbranom master rada "Fotorealistično renderovanje". Doktorske studije informatike na Prirodno-matematičkom fakultetu Univerziteta u Novom Sadu je upisao 2008. godine. Od 2008. do 2010. godine je zaposlen kao istraživač-pripravnik, a od 2010. kao asistent na istoj instituciji. Držao je vežbe za studente informatike iz više predmeta, uključujući Objektno-orijentisano programiranje 1 i 2, Računarsku grafiku 1 i 2 i Operativne sisteme 1. Bio je član nekoliko nacionalnih i međunarodnih bilateralnih projekata. Takođe, bio je ko-predsednik, sekretar i/ili član organizacionih odbora sedam međunarodnih naučnih konferencija i radionica. Gostovao je na više evropskih i svetskih univerziteta, u okviru kraktoročnih stipendija i učešća na bilateralnim projektima. Ko-autor je udžbenika iz programiranja za studente informatike, kao i skoro 30 naučnih radova iz oblasti softverskih agenata i multiagentskih sistema.

## A Short Biography

Dejan Mitrović was born on December 28, 1982 in Zrenjanin, Serbia. In 2001. he began the studies of Computer Science at the Faculty of Science, University of Novi Sad, Serbia. He graduated in 2006 by defending his BSc thesis titled "Efficient rendering of large terrains." He enrolled the MSc studies of Computer Science immediately afterwards, and at the same faculty. He defended the MSc thesis titled "Photorealistic rendering" in 2008. The same year he enrolled the PhD studies of Computer Science at the Faculty of Science, University of Novi Sad, Serbia, where he also began working as a Junior Researcher, and, from 2010, as a Teaching and Research Assistant. He conducted exercises for a number of Computer Science courses, including Object-oriented programming 1 and 2, Computer graphics 1 and 2, and Operating systems 1. He was a member of several national and international bilateral projects. In addition, he was the co-chair, the secretary and/or a member of seven international scientific conferences and workshops. Through short-term scholarships and bilateral projects, he visited a number of universities in Europe and world-wide. He is a co-author of a programming textbook for Computer Science students, and almost 30 scientific papers in the field of software agents and multiagent systems.

Novi Sad,
May 2015

Dejan Mitrović

# Univerzitet u Novom Sadu
# Prirodno-matematički fakultet
# Ključna dokumentacijska informacija

| | |
|---|---|
| Redni broj:<br>RBR | |
| Identifikacioni broj:<br>IBR | |
| Tip dokumentacije:<br>TD | Monografska dokumentacija |
| Tip zapisa:<br>TZ | Tekstualni štampani materijal |
| Vrsta rada:<br>VR | Doktorska disertacija |
| Autor:<br>AU | Dejan Mitrović |
| Mentor:<br>MN | dr Mirjana Ivanović |
| Naslov rada:<br>NR | Intelligent multiagent systems based on distributed non-axiomatic reasoning |
| Jezik publikacije:<br>JP | engleski |
| Jezik izvoda:<br>JI | srpski/engleski |
| Zemlja publikovanja:<br>ZP | Republika Srbija |
| Uže geografsko područje:<br>UGP | Vojvodina |
| Godina:<br>GO | 2015 |
| Izdavač:<br>IZ | autorski reprint |
| Mesto i adresa:<br>MA | Novi Sad, Trg D. Obradovića 4 |
| Fizički opis rada:<br>(broj poglavlja/strana/lit. citata/tabela/slika/grafika/priloga)<br>FO | 7/179/206/6/22/0/0 |
| Naučna oblast:<br>NO | Računarske nauke |
| Naučna disciplina: | Veštačka inteligencija |

ND
Predmetna odrednica/ Ključne reči: software agents, multiagent systems, artificial intelligence, artificial general intelligence, cognitive architecture, non-axiomatic logic, distributed computing
PO
UDK
Čuva se:
ČU
Važna napomena:
VN

Izvod: Agentska tehnologija predstavlja dosledan pristup razvoju distribuirane veštačke inteligencije. Ono što agente izdvaja od ostalih pristupa su autonomno, reaktivino, pro-aktivno, i socijalno ponašanje. Pored toga, kompleksniji, inteligentni agenti se često definišu koristeći ljudske mentalne konstrukcije, kao što su verovanja, želje i namere.

Disertacija se bavi softverskim agentima i multiagentskim sistemima sa nekoliko aspekata. Prvo, definisana je nova arhitektura za rasuđivanje sa primenom u razvoju inteligentnih agenata, nazvana *Distribuirani sistem za ne-aksiomatsko rasuđivanje* (eng. *Distributed Non-Axiomatic Reasoning System*) (DNARS). Umesto popularnog *BDI* modela za razvoj inteligentnih agenata (eng. *Belief-Desire-Intention*), arhitektura se zasniva na tzv. *Ne-aksiomatskoj logici*, formalizmu razvijenom u domenu veštačke opšte inteligencije. DNARS je skalabilan softverski sistem, sposoban da odgovara na pitanja i da izvodi nove zaključke na osnovu veoma velikih baza znanja, služeći pri tome veliki broj klijenata.

Zatim, u disertaciji je predložena nova multiagentska platforma nazvana *Siebog*. Siebog je zasnovan na modernim standardima za razvoj veb aplikacija, čime pokušava da smanji razliku između multiagentskih sistema i sistema koji se koriste u industriji. Kao DNARS, i Siebog je distribuiran sistem. Na serverskoj strani, Siebog se izvršava na računarskim klasterima, pružajući napredne funkcionalnosti, poput automatske distribucije agenata i otporonosti na greške. Sa klijentske strane, Siebog se izvršava u veb pretraživačima i podržava široku lepezu hardverskih i softverskih platformi.

Konačno, Siebog se oslanja na DNARS za ravoj agenata sa jedinstvenim sposobnostima za rasuđivanje.

IZ
Datum prihvatanja teme od strane NN veća: 25. april 2014.
DP
Datum odbrane:
DO

Članovi komisije:
(Naučni stepen/ime i prezime/zvanje/fakultet)

KO

| | |
|---|---|
| Predsednik: | dr Zoran Budimac, redovni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu |
| Mentor: | dr Mirjana Ivanović, redovni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu |
| Član: | dr Milan Vidaković, redovni profesor, Fakultet tehničkih nauka, Univerzitet u Novom Sadu |
| Član: | dr Kostin Badika, redovni profesor, Fakultet za automatiku, računarstvo i elektroniku, Univerzitet u Krajovi, Rumunija |

# University of Novi Sad
# Faculty of Science
# Key Words Documentation

| | |
|---|---|
| Accession number:<br>NO | |
| Identification number:<br>INO | |
| Document type:<br>DT | Monograph documentation |
| Type of record:<br>TR | Textual printed material |
| Contents code:<br>CC | Doctoral dissertation |
| Author:<br>AU | Dejan Mitrović |
| Advisor:<br>MN | Dr. Mirjana Ivanović |
| Title:<br>TI | Intelligent multiagent systems based on distributed non-axiomatic reasoning |
| Language of text:<br>LT | English |
| Language of abstract<br>LA | Serbian/English |
| Country of publication:<br>CP | Republic of Serbia |
| Locality of publication:<br>LP | Vojvodina |
| Publication year:<br>PY | 2015 |
| Publisher:<br>PU | Author's reprint |
| Publ. place:<br>PP | Novi Sad, Trg D. Obradovića 4 |
| Physical description:<br>(no. of chapters/pages/bib. refs/tables/figures/graphs/appendices)<br>PO | 7/179/206/6/22/0/0 |
| Scientific field:<br>SF | Computer Science |
| Scientific discipline: | Artificial Intelligence |

SD
Subject/Key words: software agents, multiagent systems, artificial intelligence, artificial general intelligence, cognitive architecture, non-axiomatic logic, distributed computing
SKW
UC
Holding data:
HD
Note:
N

Abstract: The agent technology represents one of the most consistent approaches to distributed artificial intelligence. Agents are characterized by autonomous, reactive, pro-active, and social behavior. In addition, more complex, intelligent agents are often defined in terms of human-like mental attitudes, such as beliefs, desires, and intentions.

This thesis deals with software agents and multiagent systems in several ways. First, it defines a new reasoning architecture for intelligent agents called *Distributed Non-Axiomatic Reasoning System* (DNARS). Instead of the popular Belief-Intention-Desire model, it uses Non-Axiomatic Logic, a formalism developed for the domain of artificial general intelligence. DNARS is highly-scalable, capable of answering questions and deriving new knowledge over large knowledge bases, while, at the same time, concurrently serving large numbers of external clients.

Secondly, the thesis proposes a novel agent runtime environment named *Siebog*. Based on the modern web and enterprise standards, Siebog tries to reduce the gap between the agent technology and industrial applications. Like DNARS, Siebog is a distributed system. Its server side runs on computer clusters and provides advanced functionalities, such as automatic agent load-balancing and fault-tolerance. The client side, on the other hand, runs inside web browsers, and supports a wide variety of hardware and software platforms.

Finally, Siebog depends on DNARS for deploying agents with unique reasoning capabilities.

AB
Accepted by Scientific Board on: April 25, 2014
AS
Defended:
DE
Dissertation Defense Board:
(Degree/first and last name/title/faculty)
DB

| | |
|---|---|
| President: | dr Zoran Budimac, full professor, Faculty of Science, University of Novi Sad |
| Advisor: | dr Mirjana Ivanović, full professor, Faculty of Science, University of Novi Sad |
| Member: | dr Milan Vidaković, full profesor, Faculty of Technical Sciences, University of Novi Sad |
| Member: | dr Costin Bădică, full professor, Faculty of Automatics, Computers and Electronics, University of Craiova, Romania |