UNIVERSITY OF OTAGO

Unifying Micro-agent Communication in the Otago Agent Platform (OPAL)

by

Christopher Frantz

A Dissertation submitted in partial fulfillment for the degree of Postgraduate Diploma of Science in Software Engineering in the Department of Information Science

October 2009
Abstract

Multi-agent frameworks are broadly available in the meanwhile. Nearly all of them however understand the agent concept differently, going as far as to provide several agent concepts within a platform. Along with this performance results of selected platforms allow to identify two clusters defined either by efficiency of processing or expressiveness of interaction language. As the only platform being related to both clusters the Otago Agent Platform (OPAL) which uses the efficiency-driven concept of 'Micro-agents' is reviewed. As a result of the analysis the current implementation of the Micro-agents is considered to be too intransparent to allow monitoring at runtime or even cross-platform interaction. As such the implementation of a unified message infrastructure is suggested to simplify development of micro-agent interaction and provide monitoring capabilities while minimizing the loss of efficiency. The integration of Clojure as functional language is further considered to provide improved facilities for concurrent processing, usability and additional agent implementation languages.

The results of the hybrid imperative-functional implementation are behind the expected outcome as the runtime takes about five times as long as the original runtime. The results still offer potential for adoption as of the usability improvements and the fact that they are still far below the runtime of the slowest platform in the comparison. The choice of Clojure as language base for the whole platform is believed to be a promising next step to improve performance and avoid switching between different programming paradigms while keeping the improvements achieved. Further research with regards to a stronger integration of concepts of Agent-Oriented Software Engineering in the amended OPAL is suggested.
Acknowledgements

I want to express my deepest gratitude to my supervisor Professor Dr. Martin Purvis, Head of the Department of Information Science, for his support in shaping this motivating research topic and sharing inspiring insights into visions going far beyond the scope of this work.
I am also deeply grateful to my supervisor Dr. Mariusz Nowostawski, Lecturer at the Department of Information Science, whose stimuli influenced substantial parts of this work. His critical review was essential for the final outcome.
A person whose understanding and will to sacrifice countless hours of togetherness made this work possible is my dearest girlfriend Nora.
## Contents

Abstract i

Acknowledgements ii

List of Figures v

List of Tables vi

Abbreviations vii

1 Introduction 1

1.1 Background & Motivation ............................................ 1  
1.2 Outline of this document ........................................... 3

2 Terminological and conceptual foundation 4

2.1 Agents ........................................................................... 4  
2.1.1 Foundations of Agent-based Computing ....................... 4  
2.1.2 Agent Communication Languages ................................. 7  
2.1.3 Agent Architectures ................................................ 10

2.2 Agent-Oriented Software Engineering ............................. 14  
2.2.1 Principles .................................................................... 14  
2.2.2 Comparison of Paradigms .......................................... 15  
2.2.3 Relevance of AOSE .................................................. 18

2.3 Comparison of Agent Platforms ..................................... 19  
2.3.1 Definition and Specifications .................................... 19  
2.3.2 Criteria ................................................................. 21  
2.3.3 Comparison ............................................................ 22

2.4 Technological developments supporting Agent-based Computing .............................................. 35  
2.4.1 Functional Programming .......................................... 35  
2.4.2 Languages for the Java Virtual Machine..................... 36  
2.4.3 Java Interoperability in Clojure ................................. 37

3 Concept, Design and Implementation 42

3.1 Current state, problem outline and hypothesis .................. 42  
3.2 Design decisions ......................................................... 47
3.2.1 Initial design ........................................ 47
3.2.2 Clojure as Directory and Message Processor .......... 50

4 Performance Optimizations and Results 56
  4.1 Performance Optimization and Testing ................... 56
  4.2 Benchmark results ....................................... 60
  4.3 Open Issues of the work ................................ 66

5 Conclusion and Outlook 68
  5.1 Conclusion ............................................... 68
  5.2 Outlook .................................................. 69

A Class diagrams ........................................... 74

B Sequence diagrams ........................................ 80

Bibliography ................................................. 83
List of Figures

2.1 Elements of a Multi-Agent System .......................... 7
2.2 FIPA Agent Management Reference Model ...................... 20
2.3 Schematic overview of Cougaar Agent with components and binders 28

3.1 Micro-agent system class diagram .............................. 44
3.2 HelloPrinter benchmark scenario ............................... 45
3.3 HelloPrinter benchmark for OPAL using performative-based interaction ........................................ 49
3.4 Architectural building blocks of the Clojure-backed OPAL 52
3.5 Dynamic discovery and binding in OPAL using Clojure ........ 54

4.1 Absolute runtimes for values up to 100,000 ..................... 63
4.2 Detail view on absolute runtimes for values up to 100,000 64
4.3 Relative performance changes for up to 100,000 iterations 65
4.4 Detail view on relative performance changes for up to 100,000 iterations ........................................ 66

A.1 Class diagram of Clojure-backed Micro-agent framework 1/5 76
A.2 Class diagram of Clojure-backed Micro-agent framework 2/5 77
A.3 Class diagram of Clojure-backed Micro-agent framework 3/5 78
A.4 Class diagram of Clojure-backed Micro-agent framework 4/5 79
A.5 Class diagram of Clojure-backed Micro-agent framework 5/5 79

B.1 Sequence Diagram of Dynamic Discovery and Binding ........ 81
B.2 Sequence Diagram of Directory functionality .................. 82
# List of Tables

2.1 Historic developments of programming paradigms .......................... 17
2.2 Synopsis Object-oriented Programming versus Agent-Oriented Pro-
gramming .................................................................................. 17
2.3 10000 'Hello World' iterations on different Agent Platforms ........ 23
2.4 Agent Platform Comparison matrix ........................................... 35

4.1 Comparison of micro-agent implementations and further platforms
(10000 'Hello World' iterations) .................................................. 61

A.1 Table of changed elements in Micro-agent framework ................. 75
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Agent Communication Language</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AID</td>
<td>Agent Identifier</td>
</tr>
<tr>
<td>AOP</td>
<td>Agent-Oriented Programming</td>
</tr>
<tr>
<td>AOSE</td>
<td>Agent-Oriented Software Engineering</td>
</tr>
<tr>
<td>B2B</td>
<td>Business-to-Business</td>
</tr>
<tr>
<td>BDI</td>
<td>Belief-Desire-Intention</td>
</tr>
<tr>
<td>CLR</td>
<td>Common Runtime Language</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>DARPA</td>
<td>Defense Advanced Research Projects Agency</td>
</tr>
<tr>
<td>FIPA</td>
<td>Foundation for Intelligent Physical Agents</td>
</tr>
<tr>
<td>GRASP</td>
<td>General Responsibility Assignment Software Patterns</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Interaction Protocol</td>
</tr>
<tr>
<td>J2ME-CDC/CDLC</td>
<td>Java 2 MicroEdition Connected Device Configuration/Connected Limited Device Configuration</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Message Service</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>KIF</td>
<td>Knowledge Interchange Format</td>
</tr>
<tr>
<td>KQML</td>
<td>Knowledge Query and Manipulation Language</td>
</tr>
<tr>
<td>KRSIL</td>
<td>Knowledge Representation Specification Language</td>
</tr>
<tr>
<td>KSE</td>
<td>Knowledge Sharing Effort</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>LISP</td>
<td>List Processing</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi-Agent Systems</td>
</tr>
<tr>
<td>MTP</td>
<td>Message Transport Protocol</td>
</tr>
<tr>
<td>OO</td>
<td>Object Orientation</td>
</tr>
<tr>
<td>OOAD</td>
<td>Object-oriented Analysis and Design</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-oriented Programming</td>
</tr>
<tr>
<td>REPL</td>
<td>Read-Eval-Print-Loop</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Socket Layer</td>
</tr>
<tr>
<td>STM</td>
<td>Software Transactional Memory</td>
</tr>
<tr>
<td>USP</td>
<td>Unique Selling Proposition</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background & Motivation

The ongoing demand to develop information systems which keep track with the ongoing technological and economical development provides a challenging task. Especially when it comes to substantial changes in existing systems in order to provide better connectivity to customers, integrate B2B processes or reflect the organizational changes caused by Mergers and Acquisitions - resulting in the demand to connect systems which were never designed to be integrated respectively interact - system complexity can rise tremendously and often ends up in hardly maintainable patchwork solutions or considerations for a reimplementation. In the past - and up to now - developments to tackle the inherent complexity of heterogeneous and distributed systems have been proposed from the research area of Distributed Artificial Intelligence which offers Multi-agent systems as a potential solution. Those systems consist of multiple agents which - put simply - are individualistic software pieces which can interact with their environment and whose underlying core principle is flexibility. As unprecise as the agent concept itself\footnote{Some researchers even advocate the lack of common understanding (see [1]). However, section 2.1 will elaborate on this.}
Introduction

are its domains of application. They range from simple applications merely coor-
dinating data sources\(^2\) over self-coordinating robust agent networks up to complex
cooperative problem solving tasks\(^3\). Key arguments for the use of agents are thus
typically their distributedness and the application in fields which are of higher
complexity than usual means of automation can tackle[4]. Along with this open-
ness is described as a central characteristic of agent systems. This leaves the
question why agent platforms providing support of standardized expressive com-
munication languages are scarcely publicly available. One answer - besides the
arguable (dis)advantages - might be that openness is yet broadly interpreted as
interoperability on the transport level. Performance might be one obvious rea-
son for this. At least observable on a spontaneous review of according platforms\(^4\)
- which by the typically large number of agents in implementations can rather
be qualified as massive multi-agent platforms - is the lack of any high-level com-
munication language support. The maintenance state of the listing of compliant
platforms for the most important standard[7] might be another indicator that
high-level standardization is not necessarily the silver bullet for a successful agent
platform.\(^5\) This however shall not lead to the interpretation that the effort put
into this was worthless. It merely indicates the matter of problem appropriateness
when turning to efficient distributed computing. As such combining the best of
two worlds is a way to go. Foundations for this are laid and this work might take
a further step on the way to provide a comprehensive approach to suit a broader
field of demands than done by existing platforms. As such this work is driven by
the idea to provide a standardized implementation of communication on the lowest
agent level to allow a systematic and transparent application of agent-oriented
development across multiple levels of agency.

\(^2\)Examples for this are agents as personal digital assistants (s. [2]).
\(^3\)A popular example for this is the well-known OASIS air traffic management system[3].
\(^4\)Here to name are Cougaar[5] and MadKit[6].
\(^5\)Many implementations are either frozen (FIPA OS since 2003[8], Tryllian since 2005[9]), not
available anymore (Grasshopper) or are vegetating outside the spotlight they had been in earlier
(Java Agent Services [10]).
1.2 Outline of this document

After providing a precision on the rather broad understanding raised above different agent types and their specializations are further elaborated. Along with this agent-related topics (e.g. communication standards) will be introduced. Following this the reader will be confronted with the basic idea of Agent-Oriented Software Engineering (AOSE) - clarifying the reason to continuously promote Agent-Orientation as a realistic developmental paradigm. Agent platforms are of central importance for this work. As such a comparison of major players is undertaken. Further foundations - here in the context of functional programming languages - yield towards the anticipation of conceptual elements used as part of the achievements of this work.

As this work covers the unification of communication on the micro-agent layer - a term yet to be introduced - of an agent platform, namely research problem, design and implementation of an approach to this, will be described and discussed in Chapter 3.

Chapter 4 is dedicated to the central matter of performance of the system and finally provides according results including the necessary discussion. Open problems respectively suggestions for improvements are given. Finally conclusions based on this approach are drawn to motivate adoption of the concept for integration into the actual framework based on its strengths but also weaknesses. Additionally further research questions related and beyond this work are identified.
Chapter 2

Terminological and conceptual foundation

2.1 Agents

Before introducing the actual problem targeted in this research first terminological foundations are to be laid. In this context the discussion of agents as the conceptual basis is inevitable. Furthermore technologies identified to be of use in this approach will be covered as well.

2.1.1 Foundations of Agent-based Computing

The fact that we have to deal with agents in our daily life in different context and different levels constitutes the necessity for a context-related introduction into this field. In the tradition of the Artificial Intelligence community the concept of an agent - and indeed typically one agent - targeted towards an at least partial representation respectively imitation of knowledge directed towards the abilities a human mind has. Along with the realization that this goal had not been reached the substantial potential for the area of computer science and specifically software engineering was identified, regarding the context of increasingly complex
systems.[12] Moving to a different field of interest the agent concept changed its shape. One broadly accepted definition for the latter notion of an (intelligent) software agent is taken from [12]¹:

’an agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.’

The author of this definition additionally elaborates the central properties in this context to give a clear idea about capabilities and limitations agents underly:

- **Autonomy**² refers to the idea that agents are able to act without direct human interaction³. This implies the control over the state they are encapsulating and the isolation of it against the environment, thus prohibiting direct influence by external affairs (events, different agents, or as aforementioned, human beings). This autonomy property further includes the agent’s potential to control its own goals; the agent per se is thus not necessarily obliged to contribute to a ‘greater good’ - or more specific for the context of Multi-agent systems - does not underly the ‘benevolent agent assumption’ as introduced by Rosenschein and Genesereth[13].⁴

- **Reactivity** is linked with the idea that an agent is situated in an environment which it can perceive (by an undefined degree of comprehensiveness and precision) and potentially react upon.

- **Pro-activeness** relates to goal-directed behaviour which allows the agent to not only react on stimuli from his environment but taking the *initiative* itself - which is in close relation to the autonomy property outlined before.

---

¹The definition is formed in [12] though the actual properties had been shaped by [11].

²The idea of autonomy must be understood in a relative way as agents typically underly given runtime constraints which typically are platform (unless being a mobile agent) and communication means.

³Here the emphasis is put on software agents; the fact that humans themselves can be agents is excluded.

⁴The concept of autonomy in agents is closely interlinked with the actual agent interpretation. A comprehensive starting point on autonomy in agents is offered by Castelfranchi [14], a more recent treatment of formal background as well as application context is provided by Nowostawski and Purvis [15].
- **Social ability** respects the fact that intelligent software agents should not only be restricted to the perception of the environment but also be capable to utilize an Agent Communication Language (ACL) allowing to interact with other agents. This interaction is not restricted to simple message exchange but potentially extends up to a knowledge-based communication level which allows cooperation and coordination between different agents in order to achieve goals beyond the capabilities of individual agents.

This enumeration of agent properties is not exhaustive but contains the central elements qualifying a piece of software as an agent.\(^5\) The definition outlined above has a so-called ”strong notion” \(^6\) of agency as of the mentalistic elements (as to be introduced for agent architectures at a later point).\(^6\) Based on this understanding a single agent will be of limited use; especially the notion of autonomy and social ability - beyond simple invocation or semantically flat request/response protocols\(^7\) - make agents attractive for interaction between different trust domains by means of cooperation, collaboration or negotiation\[^{12}\]. This leads to the demand to clarify what constitutes a Multi-Agent System (MAS). Ferber\[^{16}\] suggests that MAS are made up of an environment (\(E\)), a set of objects (\(O\)) and a subset of which qualify as agents (\(A|A \subseteq O\)) as of the non-passive nature. Additionally an assembly of relations (\(R\)) is necessary in order to link the objects. An assembly of operations (\(Op\)) finally comprises all means to use an object by agents. Operators representing action and reaction to the execution of the operations by the world (which consists at least of agents and the environment) are called the ”laws of the universe”. From this definition two extremes of agent systems can be derived: On the one hand, purely communicating MAS leave the environment respectively interaction with the environment out of scope; in purely situated MAS agents situated in an environment only interact via signals mediated via the environment

\(^5\)Another definition going beyond this scope is forged by Ferber [16]; a broad survey on different agent definitions is given by Franklin and Graesser [17].

\(^6\)The contrasting weak notion typically applies for agents focusing on lower level coordination and communication tasks like mobile agents or agents in the area of telecommunications.[19]

\(^7\)Elements and examples of Agent Communication Languages (ACL) will be discussed at a later point.
which thus only allows indirect communication. Figure 2.1 reflects these core elements of a Multi-Agent System.

Figure 2.1: Elements of a Multi-Agent System

2.1.2 Agent Communication Languages

From all preceding sections it has become clear that 'interaction matters' for the agent concept discussed here - given the system they are residing in is purely situated. Driven by the idea to resemble knowledge-based communication similar to human communication - and already implied in sophisticated abilities like cooperation and negotiation - the broadly accepted Agent Communication Languages (ACL) support this demand by adopting principles of the speech act theory formed by John R. Searle [20]. The latter underlies the idea that a set of words based on order and emphasis indicates different intentions for its use. In its simplest form the illocutionary act - a term shaped by John L. Austin [21] - supports the idea that the performance of such an act implies 'conventional consequences' like answers on question or commitments on requests. In the context of ACLs the according speech acts - like statements, questions and commands - are represented as 'performatives' in the according ACL message\(^8\). The most relevant languages of this kind are the Knowledge Query and Manipulation Language [23] and the

\(^8\)The relevant groups of 'communicative acts' to be represented in ACLs are Assertives, Directives, Commissives, Permissives, Prohibitives, Expressives. For an intuitive 'differentiation by example' of those acts in the context of agent-based computing the reader is referred to [22].
**FIPA ACL [24].**

**KQML**

The Knowledge Query and Manipulation Language has its roots in the Defense Advanced Research Projects Agency (DARPA) Knowledge Sharing Effort (KSE) Initiative and was targeting the use to share information between different knowledge-based systems. Thus the designers did not specifically focus on agents as interacting units. However the use of high-level primitives as outlined above and the fact that it is not constrained to a specific knowledge domain made it the first broadly accepted ACL. The benefits coming along with this language are simple semantics allowing comparatively fast adoption for systems of limited scope. The lack of stronger semantics in turnaround resulted in the development of numerous dialects as the implementation of stronger semantics was left to the actual developers. As a consequence establishing interoperability between different systems demands for synchronization of intended semantics and their expression in KQML. Administrative functions for service registration and networking are included in the standard and thus do not demand for expression using the actual language primitives.[25]

The actual syntax is based on a LISP-like balanced-parentheses list. An example message representing the request for printing "Hello World" is shown below:

```
(ask-one :sender UserAgent
  :receiver PrinterAgent
  :content "write ('Hello World!')"
  :ontology printer-world
  :language Prolog)
```

The actual content language being encapsulated in the ACL message is to be chosen by the developer and depends on the demanded expressiveness but also compatibility to potential external interaction partners. Typical content language candidates are - without further introduction at this point - the Knowledge Interchange Format (KIF), the Knowledge Representation Specification Language (KRSL) but also languages introduced from different fields like the Structured
Query Language (SQL).  

**FIPA ACL**

Contrasting the KQML approach the Agent Communication Language maintained by FIPA was driven by the community around agent-based computing. Based on the ACL Arcol, a development undertaken by France Telecom, it was further developed by a standardization body and thus continuously maintained. As a reaction of the lack of strength of the KQML specification the FIPA ACL is underlying a agent-focused semantic model based on multimodal logic. Its major improvement against the flat KQML model is the potential to create more complex expressions by the combination resp. encapsulation of performatives in the content language (Example: performative: inform, content: (achieve goal X)).

However, this expressiveness comes at a price: Systems implementing FIPA’s ACL are forced to implement additional simplification mechanisms like conversation policies to reflect the agent capabilities envisioned by the standard. An additional constraint which applies to systems employing the FIPA ACL is that those need to be able to support concepts from the Belief-Desire-Intention Agent Architecture to represent the reasoning implications specified in the ACL. While FIPA has its strength in expressiveness, administrative messages which are separately specified in KQML, need to be represented in the ACL and are consequently individually expressed in the actual content language.

A simple example message using the FIPA ACL, expressing the same content as above, is:

(request
 :sender UserAgent
 :receiver PrinterAgent
 :content "write ('Hello World!')")

---

9 An example for an approach to automate the translation between different content languages employing translation via an intermediate formalism based on the Chomsky hierarchy is given in [26].

10 Agent Architectures will be introduced at a later stage. For this point it is sufficient to realize that one specific architecture is assumed when using the FIPA ACL.

11 See [25] for expressions supported in KQML.

12 Complex constructs as encapsulated performatives and conversation protocols are omitted at this point.
The ACLs mentioned represent the standardized set of ACLs, which does not take the freedom of the individual developer to decide on one of those or even to implement a proprietary communication mechanism. Reasons for this could be the level of complexity of the agent system to be developed, the underlying agent architecture, efficiency of message processing and necessity for cross-platform interaction. However, support of at least one of those formalisms should be considered if the idea of distributed open systems is not to be harmed - which still is a central motivation for the take-up of agent-based systems.

Not intensively discussed but introduced at this point are Interaction Protocols (IP) or Conversation Policies which serve as specification of interaction patterns on the conversation level to overcome the pragmatic constraints agent communication underlies. Greaves et al. [27] define those as "declarative specifications that govern communications between software agents using an agent communication language." In the context of FIPA several IPs are published [28] (e.g. the Contract-Net-Protocol) to ensure interoperability for those between complying platforms.

2.1.3 Agent Architectures

After covering the basic terminology and principles of agent communication languages the internals of agents - apart from the definitorial introduction above - have not been sufficiently introduced. In the area of agent architectures many different approaches were taken most of which did not find their way into implementation or qualified as such depending on the scope of the related agent definition. Using Ferber’s understanding the term architecture, it refers to the internal organization of an agent. While, driven from this simple understanding many different approaches qualify as agent architecture\(^\text{13}\), mainly two found

\(^{13}\)Ferber provides a broad overview over numerous different architectures.
their way into mainstream agent-based computing, may it be because of strongly agent-related underlying principles, feasibility of implementation or efficiency at runtime.

**Deliberative Architectures**

The goal of agents incorporating a deliberative architecture referred to the idea to represent the theory of *Practical Reasoning* in human beings within agents. Divided into two separate processes the former one, determining *what goal to achieve* from a given set of goals, is called *deliberation*. The latter one in turnaround is named *means-end-reasoning* and answers the question on *how to achieve* the goal determined in the former process. The theoretical framework underlying this was introduced by Bratman [29], known as the *Belief-Desire-Intention* Model and found its way into the area of Computing in different approaches. This model is yet the leading paradigm for deliberative architectures; its popularity is clearly reflected by the most popular standardized ACL which assumes this model for agent implementations.

Central (philosophical) elements of the BDI architecture - are:

- *Beliefs*, which is the information the according agent has about its environment (and thus consequently about other agents) (informational stance).

- *Desires*, which include all desires of an agent. Nevertheless, desires do not necessarily lead to action but represent the general motivation (motivational stance).

- *Intentions* are the integral part of the deliberation in the BDI model. They drive means-end-reasoning (as mentioned before), typically persist (unless dropped during deliberation) and constrain future deliberation.

For further elaboration on the philosophical background the reader is referred to Bratman [29].

The first enduring transfer into the field of computing was the *Procedural Reasoning System* by Georgeff and Lansky [30]. Elements represented are

- *Knowledge Base* for storing the symbolic beliefs of the world,
• Goals to be pursued by the agent,

• Knowledge Areas respectively Plans which can be seen as a collection of actions towards the achievement of a goal,

• Intentions, which in this architecture are modeled as the execution plans currently executed and an

• Interpreter which coordinates the manipulation of each of the listed elements, thus updating beliefs, executing plans and so on.

The approach represented by this implementation and all its derivates fall under the classification as Planning Agents as the plans are the central part of architecture. The PRS system had been reimplemented by several other researchers having varying foci; distributed reasoning and time-critical execution is the key improvement of dMARS[31], Jack[32] is a commercially available approach on extending Java with BDI concepts.

PRSs are not the latest developments based on the BDI model and as a tribute to their time of emergence they often cut short on expensive deliberation due to restricted computational resources. A different approach to the BDI model with explicit representation of the central BDI elements with according logic and the stronger emphasis on the deliberation process is taken by Rao and Georgeff [33]. Later developments with regards to the BDI model concentrate on the lack of unified representation of the goal concept in terms of goal types and a declarative approach to allow reasoning about goals.15

Reactive Architectures

Another extreme opposing the state-of-the-art reasoning approach are reactive architectures which can be found in combination with the former ones but are often represented by their purest descendant, the Subsumption Architecture by Rodney Brooks[36]. The underlying principle is driven by omitting all representations of the world, following the principle “... that the world is its own best model.”[37] Instead of symbolic representation of the world and periodically updating (bringing

14 Further goal types are not distinguished in this work. For an overview and the relation to other agent-related concepts refer to [1].
15 See [34] in combination with [35].
a timely delay and lack of precision) Rodney’s approach is physically grounded: Sensors for environment perception are wired with the actuator (for manipulation of the environment by action) via behaviours or ”levels of competence” as named by the author. Those ”levels of competence” each consist of typically very simple behaviour like ”avoid contact with object” but can be stacked in order to and form the more complex overall behaviour (higher levels subsume the behaviour of lower levels). In this concrete approach the number and configuration of layers is based on the problem, not on conceptual components as in the BDI model. This results in direct reaction on environmental changes but its hard-coded nature and avoidance of a representation of its environment, lack of goals and high-level communication language keeps it from explicit reasoning and thus qualifies it as purely reactive approach.

**Hybrid Architectures**

Considering both of the approaches and the extreme examples clarifying the their characteristics it can be retraced that researchers and practitioners suggest to combine the advantages of both architectures\(^{16}\). An often-cited example is the IN-TERRAP architecture by Mueller [39] in which both architectures are layered: reactive elements for direct interaction with the environment respectively behaviour patterns are implemented on a lower level while deliberation, reasoning and planning are done on a higher overlay level. It is additionally extended by capabilities for cooperative planning which will not be discussed at this place.

More important than the examples introduced is the choice of a fitting architecture for the envisioned tasks as well as to be taken into consideration for the overall system performance as ”... there is no longer just one aim for the whole [MAS] system [but the necessity of the] agent architecture to reflect the dual role of being an individual and a community member .. [which] requires explicit reasoning about the process of co-ordination ...”[40]. In order to reflect the different parameters of coordination influenced by agent architectures\(^{17}\) Ferber\(^{18}\) provides an

---

\(^{16}\)Shen et al. [38] can serve as an example for this.

\(^{17}\)Here to name are Mode of Communication and Adaptibility as of their strong contrast within the mentioned architectures.

\(^{18}\)Refer to [16], p.410.
ordinal-measured tabular comparison to identify the adequate approach based on further characteristics.

2.2 Agent-Oriented Software Engineering

2.2.1 Principles

Given the reader reached this point, being introduced to agents, their means of communication as well as architectural foundations, it is to question why those elements provide a potential as building blocks for modern software systems. The idea complementing this is named Agent-Oriented Software Engineering (AOSE) and provides arguments for a take-up of the agent-based view on (especially complex) software systems. One key reason identified by Jennings[41] is the shift from programming languages and principles grounded in the underlying computation architecture (e.g. Assembly) towards a focus on the problem domain (e.g. object-orientation). As such Jennings agrees to the general principles used in Object-oriented Analysis and Design (OOAD)[42]:

- **Decomposition** serves to divide larger problems into smaller pieces which allow (close to) isolated treatment to reduce the developer’s scope.

- **Abstraction** reduces the level of detail the developer needs to take into account and thus concentrates the view on certain details while leaving others out of perspective.

- **Organization**\(^\text{19}\) finally concentrates on the interrelationship within the various components/elements of the system. As such grouping certain system units can support both of the principles mentioned above as groups of units may allow to be treated as a collective unit leaving out further details of the internal group structure. This implicitly supports high-level decomposition based on those groups.

\(^{19}\)This term is chosen by Jennings to avoid the wrong understanding of Booch’s *Hierarchy* in terms of control.
Respecting those elements the understanding of the units which serve for those principles differ. Here to question is if object-orientation is a suitable approach to match the problem constellation or if the problem needs to engineered to match the object. As such Jennings argues that "the agent-oriented world view is perhaps the most natural way of characterising many types of problem[s]."\[41\] To provide a stronger foundation of this statement a more grounded comparison between agents and the lowest unit of granularity of the leading development paradigm, namely objects, seems indicated.

2.2.2 Comparison of Paradigms

As such Jennings considers the objects as too fine-grained as they lose means of reflection - self-localization - and lack control over their lifetime (no thread of control) as well as actions, thus always follow a purely reactive approach which demands for a defined call hierarchy. As this demands for a prediction of all possible cases of interaction, this urges for consideration at design-time rather than run-time (as for agents). This in turnaround results in a rather fixed coupling of objects with regards to their functional dependencies and enforces strict hierarchies. Along with this comes the problem of too rigidly defined interactions which refers to the hard-coded relations in terms of invocations between objects. Especially in the context of complex systems those object characteristics harm the idea to exchange elements at runtime. Contrasting this - and omitting further explanation - the agent has the autonomy elements (own thread of control; control about its actions) and means of knowledge-level interaction which enable the loose coupling without a necessarily fully specified hierarchy.

Shifting the view from objects to agents with regards to their fitting to typical problem domains Jennings argues that agent-oriented abstractions are suitable to reflect a problem domain. The elements of agent organizations as such - including their social kind of interaction - reflect the idea of real-life organizations with individual agents acting as role-based situated problem-solvers. Their degree of
abstraction is - stronger than in the case of object-orientation - rather driven by the problem domain than programming language concepts.

With regards to flexible management of organizations - if at all - the toolset of object-orientation is restricted to design patterns supporting the explicit implementation; it is not an inherent part of the object-oriented nature. This generic aspect of agent organizations includes the idea to change agent organizations by combination or decomposition allowing to flexibly reshape agents respectively their appearance to the external perception. In the context of objects this would match the ability to change classes and types at runtime.

Along with the mentioned features it should be kept in mind that organizations are not only more flexible with regards to their structure but also agents themselves can be highly heterogeneous as of their instance-level features, meaning that the autonomous capabilities allow agents to be individualistic by learning, i.e. adding and removing behaviour or capabilities - unlike objects instantiated from a class having a common and fixed attribute structure without the ability to change this at runtime. Having this latter point in mind and combining it with the introduction on agent architectures given before agents will be driven by their beliefs and goals rather than certain attribute values as purely encapsulated state.

Undoubted however most agent frameworks and toolsets are based on object-oriented technology which clarifies that the agent-orientation is rather considering a thinking paradigm without binding it to fixed technological basis as it has grown for Object-orientation. To sharpen the discipline of implementation the term Agent-Oriented Programming (AOP)\textsuperscript{20} was coined by Shoham\cite{43}. He suggests AOP to be a specialization of Object-oriented Programming (OOP) but considers the actual implementation as "programming of mental states", thus puts the emphasis on the "mental" capabilities of agents.

To show the evaluationary development the according programming paradigms towards AOP are outlined - as thought up by van Parunak - in table 2.1. Considering the concrete use of structural elements and their equivalents in Programming a

\textsuperscript{20}At this point it shall be mentioned that the concept of Agent-Oriented Programming is preceding the AOSE idea. For sake of a top down approach in comparing Object-orientation versus Agent orientation this was considered to be acceptable.
### Terminological and Conceptual Foundations

<table>
<thead>
<tr>
<th>Machine Language</th>
<th>Structured Programming</th>
<th>Object-oriented Programming</th>
<th>Agent-oriented Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural unit</td>
<td>Program</td>
<td>Subroutine</td>
<td>Agent</td>
</tr>
<tr>
<td>Relation to previous level</td>
<td>Bounded unit of program</td>
<td>Subroutine + persistent local state</td>
<td>Object + independent thread of execution + Initiative</td>
</tr>
</tbody>
</table>

Table 2.1: Historic developments of programming paradigms

A synopsis of Object-orientation and Agent-Orientation (Table 2.2) will conclude this in-depth comparison.\(^{21}\)

However, considering the Agent-Orientation as a silver bullet for the design of software solutions is the wrong interpretation. Along with their potential come their downsides. Necessary to be mentioned as such is the necessity to clearly balance reactive and proactive behaviour. Depending on this the predictability about change within the system’s objectives might fade. Further the loose coupling results - if strongly committed to the agent principles - in non-predictable interaction patterns with regards to timeliness and interaction partner. Depending on the level of sophistication in the system the emergence of behaviour, stronger than only a change of objectives, harms the stability and reliability demanded by

\(^{21}\)This direct comparison was made up by Lind [44].
industrial-strength systems.[41]

In consequence a clear selection process is necessary to see if agents are suitable for a given problem. This might take place based on the agent feature set or the analysis of available case studies. Apart from this consideration the full process of Software Engineering - and the equivalent support available in Object-orientation - demands for appropriate methodological support. The conceptual differences outlined earlier - including the ’knowledge-level’ structure elements as well as means of interaction (see section 2.1.2) - thus resulted in the emergence of numerous different methodologies consisting of procedures for the management of the full process as well as support by graphical notations for different stages and perspectives of development. Those methodologies typically provide high-level support for all kinds of agents or constrain themselves in assuming a certain agent architecture. Well-known methodologies from this context are Gaia[45], MaSE[46], Tropos[47] and Prometheus[48] most of which have been refined over time and are have gained tool support22. For a comprehensive overview over the current state the Talveter and Sterling’s chapter on methodologies[1] should be considered. Further elaboration on this will not take place here; to understand the principles rather than the full process of development the focus on the conceptual differences shall suffice.

2.2.3 Relevance of AOSE

Keeping the time of emergence of the concept of AOSE in mind - along with the judgement about the failure of the AI community (to deliver robots with human-quality minds) by one of the key researchers[11] - it is to argue whether they themselves failed to keep the promises to introduce agent-oriented computing respectively AOSE as prevalent development paradigm for complex systems, as lately done by key figures of the community[50]. Nevertheless, the critics can be seen from a motivational perspective some being driven by a market-based view as well as a conceptual viewpoint. One of those (conceptual problems) refers to the

22Here to mention is the Prometheus Development Tool[49].
level of abstraction of agents to support multiple levels in order to better reflect the different granularities for process design on the low level - and will be addressed in the later part of this work.

Even if the agent community has not been able 'to get the message across' the existing field of - especially object-oriented - software engineering still suffers from issues which qualify agents as means to address those. A good indicator for this is the field of design patterns many of which drive the idea of object-orientation to an over-engineered composition of best practices opposing the idea of simplicity in agile software development.\textsuperscript{[51]} An example showing that the demands put upon object-oriented development exceed its conceptual scope is the collection of the \textit{General Responsibility Assignment Software Patterns} (GRASP)\textsuperscript{[52]}. Main intention of the included "design patterns" is to ground the modeling of real-life problems in the according structures of object-orientation and as such better serve an understanding as an reinforcement of (OO) "design principles". Being appropriate or not, several of the patterns mentioned could be directly addressed when switching to an agent-oriented development perspective.\textsuperscript{23} The latter points, being of qualitative nature though - along with the technological changes and the arising potential for the use of agents even in small-scale systems as to be shown in section 2.4 - motivate to put continuous effort into the development and promotion of AOSE.

\section*{2.3 \hspace{1em} Comparison of Agent Platforms}

\subsection*{2.3.1 \hspace{1em} Definition and Specifications}

Being aware about the principles underlying agents and their potential advantage for the use for the construction of (complex) distributed systems one building block for implementation is the choice of an agent platform (or the decision to develop an individual solution). Thus this section will focus on several selected agent

\footnote{A non-exhaustive selection of the ones potentially substituted by shifting to an agent-oriented perspective is \textit{Information Expert}, \textit{Low Coupling}, \textit{High Cohesion} and \textit{Indirection}.}
platforms, outline their key features and potentials to provide a brief evaluation and situate the existing OPAL platform. For this purpose we consider a subset of the FIPA definition of an agent platform as suitable: "An Agent Platform (AP) provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system, agent support software, .. agent management components ... and agents."[53] Along with this the FIPA provides a Reference model for agent management which will be briefly outlined and refers - if not noted differently - to [53]. The FIPA Agent Management Reference Model

![FIPA Agent Management Reference Model](image)

**Figure 2.2:** FIPA Agent Management Reference Model

as shown in Figure 2.2 is a logical model for components to be included into Agent Platforms which seek for compliance with the reference model as part of the idea to provide interoperability between different platforms; the high-level communication demands for common understanding of the underlying platform (and reflected in the fipa-agent-management ontology). As such the FIPA reduces necessary elements of agents in this context to its existence as computational process implementing autonomy as well as being a communicative facility of an application. As prerequisite for this the agent must provide one or more services, have at least one owner and a notion of identity - the so-called Agent Identifier (AID).

The Agent Management System (AMS) controls the access and the use of an Agent Platform. The component needs to be present in a FIPA-compliant system and
serves as central directory for agents and assigns the AID upon registration.
The presence of *Directory Facilitator (DF)* however is not necessary on an agent platform. It serves as a yellow-page service for agents. Multiple instances of it can reside on the same platform.
The elements mentioned are backed by the *Message Transport System (MTS)* of the platform which provides the default means of communication. It is the lowest level of communication specification provided by FIPA.
This high-level overview of a Reference architecture for an agent platform helps to understand typical capabilities. Even if not explicitly complying to the FIPA specifications, the functionality can often be found implicitly in Agent Platforms and is thus worth to be considered as background for the understanding.
Apart from the terminological and conceptual foundations of Agent Platforms a short overview of several agent platform implementations shall be provided, partially to support evaluation as well as to get an idea about the field of functionality the platform of interest, the Otago Agent Platform, is ranging in.

### 2.3.2 Criteria

In order to pursue a comparison between several agent platforms it seems inevitable to provide central criteria based on which qualitative judgement can be undertaken. Nevertheless, independent from this a short SWOT analysis is provided for each platform to help the identification of platform potentials. Agents as building blocks for applications based on the according platform define the level of granularity the developer has to handle and thus are important when considering the area of application and the concepts agents need to represent. Thus the *Agent concept* in the according platform will be taken into account. The qualitative measure for this is based on the strength of the agent notion in the according concepts along with capabilities the platform provides for the individual agents.
The second characteristic of interest are the *Means of Communication* provided out of the box. Here not only the actual available technologies but also the relation to any agent-related communication standards is of relevance.
Another criterion are the *Aspects of Reasoning* provided by the platform respectively the community around it. Aspects of reasoning in this context refer to the integration of reasoning engines, be it in number or sophistication.

Last but certainly not least in the list of attributes is the matter of *Performance*. As this analysis does not leave space for exhaustive testing respecting all potential features for the according platforms a basic run of a 'Hello World' benchmark is conducted for (nearly) all platforms to provide an idea about the performance area they are ranging in.

As the criteria do not neutrally reflect the 'USPs' of the according platforms an outline of the Strengths, Weaknesses, Opportunities and Threats is given after each description of the according attributes.

### 2.3.3 Comparison

**Java Agent DEvelopment Framework (JADE)**

JADE is arguable one of the most used agent platforms in the open-source community respectively considered as the leading FIPA-compliant agent framework. Its origin reaches back to 1998 as an project of Telecom Italia, driven by the need to provide a validation of the FIPA specifications and after partial funding by the European Commission reached the state of a matured middleware platform. In 2000 JADE went finally open-source.[54][24]

**Agent concept**

In JADE agents are interpreted as autonomous (own thread of control and life-cycle management) and proactive. The interpretation of an agent in the JADE platform thus tends towards the strong agent notion. Along with JADE’s notion of *behaviours* agents can be extended by individual code respectively extension of the modular platform.[25] Agents can further be partially (only agent code) or fully

---

[24] JADE, now in version 3.7, is publicly available under [55]. If not cited differently information on JADE is taken from [54].

[25] In the third iteration the JADE structure was fully modularized.
(including state) mobile. This is complemented by the Location-transparent API for asynchronous messaging.

**Means of Communication**

Additionally JADE agents are assumed to be communicative and support full communication according to the FIPA specifications. Intra- as well as inter-platform communication takes place in a fully FIPA-compliant manner.

**Reasoning**

Reasoning of JADE agents does not underly a certain architecture and as of the default implementation can be considered as reactive. No supplementing reasoning engine is provided - despite the necessity of the BDI concept for the FIPA implementation. Here this is obviously left to the individual developer. Nevertheless, JADE is able to harvest out of its community: A BDI implementation specifically for JADE named JADEX is available the fill up this gap.[56]

**Performance**

Performance is the downside of JADE partially related to the heavy-weight FIPA implementation. This overhead gets apparent when performing tests like the ‘Hello World’ benchmark originally written by Mariusz Nowostawski[57] and adapted to the current JADE version. The results for all available tests are shown in the table below (Table 2.3) and give a tentative idea about the performance but will receive a more in-depth discussion in section 4. Without further discussion at this point the performance is considerably worse than any of the competing platforms.

<table>
<thead>
<tr>
<th>Agent platform</th>
<th>Runtime in Milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>JADE</td>
<td>6781</td>
</tr>
<tr>
<td>MadKit</td>
<td>781</td>
</tr>
<tr>
<td>OPAL Micro-agents</td>
<td>170</td>
</tr>
</tbody>
</table>

Table 2.3: 10000 ‘Hello World’ iterations on different Agent Platforms

The remaining features can be populated into a SWOT scheme as following:

- **Strengths**

---

26 All tests were left original as far as possible, given the changes in the underlying platforms (JADE and MadKit). All tests were run on an Intel Core2Duo 2.4Ghz machine with 3GB RAM under Microsoft Windows XP Professional with JVM 1.6.0.12.
– Modular distributed Platform
– full FIPA compliance
– Support for agent mobility
– Monitoring and Debugging toolset
– Support for J2ME-CDC/CDLC
– Interaction protocol library
– strong GUI support for management and runtime analysis

• Weaknesses
  – only slow heavy-weight FIPA-based communication
  – focus on Java as implementation language
  – no in-built high-level reasoning engine

• Opportunities
  – Maturity of platform encourages industrial uptake
  – active developer/user community providing support and extensions (e.g. Jadex)
  – open source and user community reduce cost and risk for use
  – Implementation support by Consulting companies (e.g. Whitestein AG)

• Threats
  – decreasing relevance of FIPA standards in performance-oriented massively communicating MAS

MadKit
Another platform with different scope is MadKit which had been majorly driven by Jacques Ferber and Olivier Gutknecht of the Université de Montpellier II. It follows the Agent/Group/Role model\textsuperscript{27} which defines the organization as a "framework for

\textsuperscript{27}The meta-model description is given in [58].
activity and interaction”[6]. An organization consists of groups consisting of roles which are handled by agents. Nevertheless, unit of modeling is the organization; an agent-oriented view is explicitly avoided. As such the framework concentrates on the organization of structural relationships between agents. In consequence the authors entitle their framework as multi multi-agent system.[6] Part of this idea is to have one representative agent which handles all extra-organization communication and thus hides the actual organization while avoiding superfluous communication. Based on this concept MadKit qualifies as platform for heavily distributed applications and presumably provides a strong scalability properties.

**Agent concept**

Considering the model underlying the agent platform the agent in MadKit is specified as a active communicating entity which handles roles in groups. Apart from this the interpretation is left to the implementor, thus leaving a maximum of freedom to the developer. Following this the general idea of an agent tends towards the weak notion (no implication on proactivity, lifecycle control may be taken over by organization, agents implicitly of small size) while the organization gets closer to the idea of the ‘stronger’ definition as outlined before.

**Means of Communication**

Agents in MadKit communicate via asynchronous message passing either directly to an agent address or a higher-level role-based message delivery mechanism. In the implementation the communicator or, according to the more recent specification, netcomm agents use socket-based network transport but may also implement an open set of protocols to be integrated into the connection handling (e.g. CORBA). Per se no knowledge-level communication is involved; FIPA or KQML specifications are not reflected.

**Reasoning**

As the agent concept has a high degree of freedom in implementation (size, implementation languages) no direct reasoning support is provided. As of the nature of the platform to provide a huge number of light-weight agents the platform does

---

28This is not tested in this context.
29As of lacking documentation this assertion is based on review of the actual implementation.
30The authors are planning to include FIPA support in future version.[59]
not target advanced reasoning but leaves it to the implementor. In consistence with the concept a high-level-reasoning agent would reside within one organization (not per agent), keeping the organization as main level of granularity.

**Performance**

Performance can be seen as the key advantage of the platform. Cheap communication and light-weight agents, while organizing a clear message flow within each organization, are the major enablers for this. The ‘Hello World’ benchmark - as seen in Table 2.3 - shows this performance.

Some of the features outlined and the ones omitted yet are summarized in the SWOTs:

- **Strengths**
  - light-weight multi-level platform concept based on the AGR model
  - open agent concept (e.g. scheduled or thread-based agents)
  - strongly distributed micro-kernels
  - support for Java, Scheme (Kawa), Jess, Jython and Beanshell, extensible for further script languages
  - high performance
  - ‘easy-to-use’ framework with high versatility (e.g. industrial-strength systems, simulations)
  - open source solution

- **Weaknesses**
  - no FIPA support, only low-level communication protocols
  - no reasoning engine included
  - lack of documentation (outdated and incomplete, API documentation partially only in French)
  - no real community outside of research group
• **Opportunities**

  - increasing versatility of platform by extending with high-level agent concepts and knowledge-level communication (implicitly by FIPA support)
  
  - agent migration support envisioned
  
  - IDE envisioned by authors

• **Threats**

  - FIPA support assumed to be limited (probably to ACL) as of current platform structure
  
  - development only driven by research group (risk of ceasing development)
  
  - lack of support for developers, lack of documentation for new features limits uptake outside of research group

---

**Cognitive Agent Architecture (Cougaar)**

Another platform for consideration - and not yet covered in the context of the selected platforms - is Cougaar. This platform is the result of a DARPA research project whose major goal was to provide a highly scalable agent-based platform for distributed systems of any size. A second key objective is to ensure high survivability of the system. Especially the latter characteristic indicates the relation to military logistics demands. After completion of the original research project the platform went open source but is still majorly driven by the original main contractor BBN Technologies[60]. However, up to today the platforms finds its major use in military systems[61]. The related fact that the platform both focuses on robustness and scalability and has a growing community[62] makes it worth to be considered in this comparison.

**Agent concept**

In Cougaar an agent is a software entity which autonomously communicates with
other agents to serve domain-specific functionality. This rather broad understanding wraps the architectural approach in which components provided by the platform for core functionality (like agent-2-agent messaging, naming, mobility, UIs) as well as components written by the implementers are ’plugged into’ standard agents which serve as runtime containers allowing interaction between components via the blackboard principle.\footnote{\textsuperscript{31} These agents can be enhanced by additional (cross-cutting) binders which, if chosen to apply, wrap the individual agent components.\textsuperscript{32} Examples for this are binders for security to prohibit policy violation even within agents. Figure 2.3 provides a schematic overview of a potential agent. This brief idea about the architecture shows the degree of complexity the platform is designed to handle reliably.\textsuperscript{33}}

\begin{figure}[h]
\centering
\includegraphics[width=0.6\textwidth]{cougaar-agent-diagram.png}
\caption{Schematic overview of Cougaar Agent with components and binders}
\end{figure}

\textbf{Means of communication}

Communication in Cougaar is realized as part of a MessageTransport layer which provides 'link protocols' such as loopback in-memory transport for internal communication, (SSL-supported) RMI, CORBA, HTTP, SOAP and JMS. Additional\footnote{\textsuperscript{31} An interesting feature is provide servlets for any agent to provide easily decentralized accessibility for runtime management/analysis. \textsuperscript{32}From the conceptual point of view this can be considered similar to the idea of separation of function from cross-cutting concerns in Aspect-Oriented Programming. \textsuperscript{33}To get a real overview on the platform architecture the review of the Cougaar Architecture Document\textsuperscript{[63]} - albeit being slightly outdated - is indispensible.}
protocols can be plugged in; socket programming is a further option supported. As means of serialization Cougaar relies on the native Java implementation.[65] Agent communication languages of higher level are not directly supported as this collides with the key focus on high performance and scalability.\textsuperscript{34}

**Reasoning**

Despite the fact that the full name of the platform implies cognitive capabilities this is not a central characteristic of the platform; the ‘planning domain’ supports tasks, workflows and schedules as data structures. Those are communicated via the blackboard functionality of the system to ensure consistency throughout the system (remove dependent task and task deletion, ...). A ‘planning clock’ serves as a central mechanism for planning execution by notifying components. Alternatively the developer can amend those mechanisms or introduce own ones. The planning capabilities provided out of the box serve as workflow management mechanisms - especially concentrated on the logistics domain - and rather focus on reliable and efficient execution and in consequence leave stronger deliberative concepts out of scope. Later refactoring reflects the relatively limited relevance as the use of the planning domain and the according libraries is not mandatory anymore.\textsuperscript{35}

**Performance**

In Cougaar a central objective is the ability to scale to any size application logic allows (from PDA to fully integrated military logistics system). Considering the idea to put only application logic as constraint for the size of the actual system and facing Cougaar with hierarchical decomposition of complex tasks, coordinating over thousands of different organizations while dynamically maintaining plans at execution time inherently raises performance demands. As such Cougaar reduces the idea of loose-coupling by heavily relying on typically fast transport protocols leaving the heavy-weight ACL approaches out of scope (see above). Along with the scalability the other key objective of survivability would be harmed by high-level

\textsuperscript{34}To the authors’ knowledge no heavily distributed application comparable to their Cougaar implementation relied on ACL standards like FIPA and KQML.[5]

\textsuperscript{35}See [66] in combination with pattern descriptions in [63].
communication as the developers rely on the modeling in terms of emergent be-
behaviour derived from interaction between (relatively) low-level agents (see above).
However, the performance is still target for evaluation and at the time of writing of
this work no HelloWorld benchmark for Cougaar had been written by the author.
Thus the quantitative evaluation is omitted for this platform.
Summing up the analysis and pointing out key features a SWOT overview for
Cougaar is as follows:

• **Strengths**
  
  – versatile platform with strong domain-specific elements for logistics use
  
  – key focus on scalability and ‘survivability’
  
  – numerous architectural features out-of-the-box
  
  – Cougaar is methodology and software
  
  – proved in practical use (military logistics), sample implementation avail-
    able
  
  – agent mobility support
  
  – IDE plug-in for Eclipse available

• **Weaknesses**
  
  – no strong reasoning support
  
  – no standardized ACL support
  
  – learning of the complex architectural model may only pay off in suffi-
    ciently large implementations

• **Opportunities**
  
  – constantly ongoing development
  
  – (close to) up-to-date documentation
  
  – open-source software with increasing community
Terminological and conceptual foundations

– development backed by original contractor minimizes risk of decease
– key areas of applicability identified in documentation

• Threats

– consistent integration of knowledge-level interaction could exceed capabilities of agent concept
– large architecture needs to be consistently developed (hard for single developer contribution)
– interaction with other platform not considered; if so, brings risk of breaking strong security features of platform

Otago Agent Platform (OPAL)
The last platform in this evaluation is the Otago Agent Platform (OPAL) which had been developed at the University of Otago. It is a multi-level platform driven by the idea to pursue a consequently agent-oriented decomposition to model distributed applications[67]. Focus is to provide a pure interpretation of the AOSE approach (which has been outlined in previous sections) without providing a fully-fledged heavy framework as available with some of the platforms mentioned earlier. One of its key qualities is the full compliance with the latest FIPA specifications. Its modularity qualifies it for flexible uses and it has been extended in various works some of which are the integration of coloured petri-nets for the specification of interaction protocols[68], an extension to specify those via Agent UML[69], a modular conversation manager[67] and last but not least support for different high-level reasoning engines[70].

Agent concept
OPAL consists of several levels of agents which are differentiated by their degree of complexity. Agents are interpreted as "an entity deployed in a multi-agent system"[67] which plays one or more roles in the agent society. Opposing this high level interpretation of agents micro-agents represent "the lowest and most primitive level of agent instantiation"[67]. Both concepts differ in qualities of
communication, ability to perform reasoning tasks and their autonomy. As such micro-agents do not necessarily have their own thread of control and can even be reduced to pure responsiveness and as such being merely invoked by other agents. The concept is not restricted to this two-level approach but allows to build a finite tree hierarchy rooted in the system owner agent whose internal nodes are non-primitive micro-agents in contrast to the primitive ones residing at the external nodes. Bearing this concept in mind an application can be purely composed of agents respectively micro-agents.\textsuperscript{36}

**Means of communication**

A central feature of OPAL is the support of the FIPA ACL specification for agent communication along with its implements of the necessary components of the FIPA Abstract Architecture respectively Agent Management Reference model (as outlined in section 2.3.1). OPAL additionally provides all basic Java Agent Services (JAS)\textsuperscript{37} and supports RMI, FIPA 2000 HTTP and IIOP as Message Transport Protocols (MTP).\textsuperscript{72} All this yet accounts for the high-level agents within OPAL; the micro-agents as of current state do not support cross-platform interaction unless manually implemented by the developer.

**Reasoning**

Given the feature set mentioned up to now OPAL is qualified to pursue advanced reasoning tasks by being open for the use with different high-level reasoning engines\textsuperscript{70}, thus supporting more sophisticated functions relating to the 'strong' agent notion. However, the micro-agents do not provide such advanced functionality.

**Performance**

OPAL supports the idea of AOSE which is reason of its approach of hierarchical decomposition. A further reason is indeed the ability to provide fast communication between agents. The first implementation of micro-agents achieved performance results outrunning all of the other platforms tested (see Table 2.3). However, micro-agents do not cross platform boundaries and in consequence rely on FIPA communication of higher-level agents for this purpose.

\textsuperscript{36}For the original concept refer to [71] and [67].

\textsuperscript{37}For details on JAS refer to the documentation provided along with the code at [10].
• **Strengths**
  
  – modular platform with tested support of the latest FIPA specifications
  
  – strong multi-level agent concept with different qualities (agents, micro-agents)
  
  – standard-based communication (FIPA, MTPs)
  
  – open source software

• **Weaknesses**
  
  – limited community and use outside research group
  
  – no IDE/GUI support for development
  
  – freedom of micro-agent development (implementation, communication) too high (no documentation apart from initial publication)
  
  – interaction between micro-agents not monitored

• **Opportunities**
  
  – unique approach of strictly agent-oriented development
  
  – micro-agents with cross-platform interaction would enable different fields of application (distributed micro-agents)
  
  – integration of high-level reasoning engines proven

• **Threats**
  
  – independently created platform modules may harm coherent structure of the system

**Summary**

The short evaluation of those different agent platforms shows parallels and differences. One characteristic shared by all platforms is the use of Java as their base, which, in the context of the agent concept and its focus on problems rather than
platforms, seems to be a rather obvious choice as of the platform-independence\textsuperscript{38} of the language. However, the actual comparison has more interesting findings to offer:

The platforms compared can be clustered by different dimensions which are, broadly speaking the ones using standardized ACLs like JADE and OPAL versus the ones only supporting fast communication on the message transport level. The use of advanced reasoning capabilities correlates with this observation; in Cougaar and MadKit the typical agent function is to provide a runtime container for composable functionality. The degree of distribution and complexity those systems can achieve largely differs; the successful use of Cougaar in heavily distributed systems with advanced demands on robustness and security is beyond the scope of any other known implementation. MadKit serves as an alternative of academic offspring with a stronger conceptually embedded agent understanding. JADE however is the best-known representative of agent-based platforms but primarily focused on comparatively slow FIPA-based communication which is certainly a central reason for the poor performance in the heavily interaction-based benchmark scenario\textsuperscript{39}. OPAL finally has the potential to break the cluster boundaries as its conceptual strength lies within its multi-level infrastructure which allows the use of fully FIPA-compliant communication and advanced reasoning capabilities as well as efficiency-driven low-level communication. However, the two levels of interaction in OPAL need to be clearly identified and interaction needs to be made explicit and retraceable to support the developer with the necessary transparency. The following table (Table 2.4) provides an high-level summary of the identified dimensions for the individual platforms.

\textsuperscript{38}The word platform in this context needs to be treated as Operating System.

\textsuperscript{39}The scenario has not been explicitly outlined in this document yet. Only a reference to the original source was given above. The explicit description will take place at a later point in context of the presentation of results from this work.
### 2.4 Technological developments supporting Agent-based Computing

#### 2.4.1 Functional Programming

After providing the agent-related terminological foundation for this work a short overview on another programming language paradigm is given to anticipate considerations for the work described in the upcoming chapter.

**Functional Programming** is rooted in the $\lambda$-calculus and as such treats the evaluation of functions to express a program. Key characteristics are the use of higher-order functions which are used as input, means of computation and output, avoidance of (mutable) state and side-effects and finally the strong use of recursion. Functional languages can in consequence reduce the complexity of programs and
improving the readability. Given a set of functions for composition this approach may result to be less error-prone as no mutable state has to be maintained. This implicitly qualifies according languages for the use in the context of concurrent processing.[73]

As of the immutability feature and the consumed runtime memory when heavily used 'laziness' is a key mean to reduce the use of memory as functions marked as lazy are only evaluated when explicitly demanded\(^{40}\). This feature can be found in programming languages of any paradigm but its relevance in the context of functional programming languages makes it worth mentioning.

### 2.4.2 Languages for the Java Virtual Machine

Along with the revival of this 'old' paradigm a trend of re-implementating languages in a platform-independent manner, often based on the broadly accepted and available Java Virtual Machine (JVM), can be perceived. Two notable high-profile languages - in contrast to scripting approaches - are Scala and lately Clojure. 

**Scala**[75] seeks to be multi-paradigmatic in that it supports object-oriented programming along with the advantages of functional programming. However, as every value in Scala is an object it can also be described as a pure object-oriented language with functional extension. It has an expressive static type system but can be easily extended. From Scala Java functionality can be directly accessed and the general syntax is very close to the Java’s in order to ease the adoption. Interoperation with .NET is ensured by Scala’s support for the *Common Runtime Language (CLR)*. Scala not only tries to combine the two paradigms and both relevant software platforms but also adopts concepts and advanced functionality from other object-oriented programming languages (e.g. uniform object model from Smalltalk, nestable classes from Beta) in a best of breed approach. A comparing overview over Scala and other languages is provided by Odersky et al.[76].

\(^{40}\) An example for lazy execution in Scheme is shown in [74].
is its website.[75]

Clojure\textsuperscript{41}, as a recent example of a JVM language, is a LISP\textsuperscript{42} dialect developed by Rich Hickey. It is considered to introduce the powerful functional features outlined in the beginning as well as providing dynamic typing (in contrast to Scala). From LISP Clojure inherits its homoiconicity easing the manipulation or creation of new functions by functions. Clojure also provides LISP’s Read-Eval-Print-Loop (REPL) to support interactive programming. As such the Clojure is not compiled at development but runtime\textsuperscript{43} and does not demand for a separate compiler like Scala does. In tradition of LISP Clojure code is expressed as Symbolic expressions respectively \textit{S-expressions} which are either atoms or recursively decompose atoms (as base unit of manipulation) into head and tail component.\textsuperscript{44} Clojure can directly access Java functions enabling to use any library imported in similar fashion as done in Java. Thus any Java functionality can be used/expressed in Clojure. However, the pure functional approach\textsuperscript{45} is harmed when handling Java objects as of their mutability - especially when considering concurrent processing. Clojure addresses this by introducing more powerful concepts than the lock-based Java approach. Here to name is \textit{Software Transactional Memory} (STM) as well as the constructs of atoms and agents.[81] A helpful comprehensive resource for developing applications in Clojure is [82].

\section*{2.4.3 Java Interoperability in Clojure}

As the direct interoperation with Java is a key feature of Clojure this functionality is outlined in this subsection. The fact that this information is not comprehensively provided at one place but several sources makes it worth compiling in a separate section.

\textsuperscript{41}Source for the information about Clojure - if not cited differently - is the Clojure Online Reference [77].

\textsuperscript{42}The name LISP is derived from the original function as List Processing Language.

\textsuperscript{43}Even if it sometimes appears hard to differentiate between those when considering the interactive programming potential.

\textsuperscript{44}For a definition of S-expressions in LISP context please refer to [78]. The only though unsuccessful standardization approach was taken by Rivest[79].

\textsuperscript{45}If true for LISPs anyway (see [80]).
For interoperability three main cases have to be differentiated and will be elaborated:

- Accessing Java from Clojure,
- Running Clojure code from Java and
- Invoking Clojure functions from Java.

**Accessing Java objects from Clojure** is the simplest approach. Clojure differentiates into access of static methods and instance methods. Access to static methods is achieved in the pattern `(Classname/staticMethod <arguments>)`. Examples for this executed in the REPL would be ('->' represents system return values)

```clojure
(Thread/currentThread)  
-> #<Thread Thread[REPL Thread,5,main]>  
(String/valueOf (Math/PI))  
-> "3.141592653589793"
```

For the access of instance members for classes or objects the notation is consequently:

```clojure
(.instanceMember Classname <arguments>)
```

respectively

```clojure
(.instanceMember instance <arguments>)
```

The '.' (dot special form) indicates that Java functionality is accessed. A simple example from this category is

```clojure
(.substring "Clojure" 0 4)  
-> "Cloj"
```
An example derived from the context of this work is

\[ (.\text{receive} (.\text{getReceiver} \text{message}) \text{message}) \]

which calls receive(message) on the receiver defined within the message (.getReceiver message).\textsuperscript{46}

**Running Clojure code from Java** is a second central case. The basic idea behind this is to construct a Clojure command which should be executed in a Clojure instance. The return values are usable in Java.

This functionality relies on three elements:

- The *Namespace* the command should be executed in. This may be of relevance to ensure the availability of functions only available in the according Clojure namespace.

  This can be passed as string which is then to be internalized as a Clojure symbol and either resolved to an existing namespace or newly created:

  \[
  \text{Symbol nssym} = \text{Symbol.intern(namespace)};
  \text{ns} = \text{Namespace.findOrCreate(nssym)};
  \]

- The *Bindings* of input variables need to be loaded into the namespace. To achieve this, the bindings (in this example passed as a map structure) are iterated and saved to a Clojure Associative data structure - here called mappings:

  \[
  \text{Associative mappings} = \text{PersistentHashMap.EMPTY};
  \text{mappings} = \text{mappings.assoc(RT.CURRENT_NS, ns)};
  \text{Iterator<String>} \text{iter} = \text{bindings.keySet().iterator()};
  \text{while (iter.hasNext())} \{
      \text{String key} = \text{iter.next().toString()};
      \text{Symbol sym} = \text{Symbol.intern(key)};
      \text{Var var} = \text{Var.intern(ns, sym)};
  \}
  \]

\textsuperscript{46}Main source for this approach is [83] which provides further details in this context.
Object value = bindings.get(key);
mappings = mappings.assoc(var, value);

• Finally the thread bindings are established, the command (passed as string (Variable name 'script' in the example)) is executed. The return value is of type Object and can then be treated according to the implementing function. After executing the thread bindings should be removed to avoid interference with old bindings - in case of succeeding executions of Clojure code.

Var.pushThreadBindings(mappings);
Object ret = Compiler.load(new StringReader(script));
Var.popThreadBindings();

As it can be seen a considerable processing overhead is necessary to process a line of Clojure code. The alternative is to provide a running instance and the according bindings to improve the performance or share state between different executions.\(^\text{47}\)

**Invoking Clojure functions from Java** refers to the direct invocation of existing Clojure functions and keeping this instance for further execution.\(^\text{48}\) In order to achieve this first a script with developed Clojure functions ("CustomFunctions.clj" in the example) can be loaded into a Runtime instance.

try {
    RT.loadResourceScript("CustomFunctions.clj");
} catch (Exception e) {
    e.printStackTrace();
}

Next the according Clojure function ("get-agent-for-role") from a given namespace (ns as String) is bound to a Var which is accessible by Java:

Var getAgentForRole = RT.var(ns,"get-agent-for-role");

\(^\text{47}\)Source for this approach has been a discussion with Mariusz Nowostawski. Further information can be found in the Google Group for Clojure [84].

\(^\text{48}\)This functionality relies on the imports of clojure.lang.RT and clojure.lang.Var.
To access the actual function Java needs to invoke it with the according parameters:

```java
try{
    Object res = getAgentForRole.invoke(role);
} catch (Exception e){
    e.printStackTrace();
}
```

The result can then be processed by the succeeding Java code. As long as the reference to the RT instance (Clojure Runtime instance) is kept it is repeatedly accessible and thus a way to share state between different invoking entities. Even if the delivered explanation is rather detailed it might help to understand design decisions taken at a later point.\(^{49}\)

\(^{49}\)Source for this approach is [85].
Chapter 3

Concept, Design and Implementation

3.1 Current state, problem outline and hypothesis

After discussing the foundations of agents, their means of communication and a brief comparison of agent platforms this chapter concentrates on the actual research problem as a contributing part to earlier work.

As outlined in the beginning and during the comparison between several agent platforms the Otago Agent Platform (OPAL) has the powerful concept of decomposing high-level agents - from hereon just called agents - from micro-agents in a finite tree structure.\(^1\) Micro-agents residing on internal nodes, and thus implicitly "own"\(^2\) the subjacent nodes, are called non-primitive micro-agents; micro-agents on external nodes are primitive ones. This way it can be ensured that no agent resides outside this hierarchy.\(^3\) As such the organization model of OPAL is strictly

---

1 Even if it has been introduced earlier a more intensive treatment seems necessary at this point to clarify the differences between the different agent types to understand design decisions. The conceptual source for this section - of not marked differently - is [71] and [67].

2 They are in fact the owners of the groups their subjacent agents are loaded into.

3 In the extreme case a primitive micro-agent is child of the agent being system owner; the system owner is not owned by any other agent.
hierarchical. Parent nodes on the micro-agent level can be in a control relationship with their child nodes. They decide about delegation which as such limits the autonomy of the nested micro-agents. This ensures a clear system structure but per se restricts direct interaction of micro-agents beyond their own branch. However, the discovery mechanism should ensure that every capability is usable via its parent micro-agent. Other models of choice would be a market of self-controlled capabilities or a network which would be driven by request and/or delegation of capabilities between equal peers. For further details on those different organization models Sterling and Taveter’s section on Roles, Goals and Organizations[1] is worth to be consulted.

The Micro-agent system as such contains three logical key concepts first of which are the Agents as actors. They play one or more Roles and each agent is assigned to one Group. As groups are roles they are 'played' by their according agents acting as group owners. In this way the hierarchy outlined before is kept consistent. In order to clearly separate micro-agents from their coarse-grained high-level relatives the central differences are outlined at this place:

Micro-agents are implemented at the base of the platform with closest binding to machine level. They may exhibit social roles, a specialization of roles and interact via direct method calls, thus avoiding all the processing-intensive elements of standardized ACLs, explicit use of ontologies, interaction protocols and parallel communication with several agents. In order to resemble the spirit of speech acts for conversations methods for interaction (introduced by the Social Role) are named after according performatives. Parameters for each method are the instance of the sender and the goal instance. This reflects the disability in Java to identify the caller of a method. Figure 3.1 shows the class diagram summarizing the concept.

4The term of 'actor' from the original document should be treated with care as the abilities of micro-agents can go beyond that of the actor concept. A comparison between actors and agents is provided by Ferber [16].

5It should be noted that the Ferber’s concept of interpreting organizations as the result of lower-level components is similar (and is used as the AGR model in MadKit). He interpretes agents as a special case of organizations but considers those to be built out of modules rather than agents (and from his understanding organizations)[16].
Modification of diagram from [67]

**Figure 3.1:** Micro-agent system class diagram

Based on these simple concepts the authors achieved promising performance results — as to be seen in Table 2.3 in context of the platform comparison. In this context it seems adequate to give an idea about the benchmark scenario used in a yet unintroduced manner. It consists of a User Agent - in fact being a primitive micro-agent - which requests a printing service from a HelloPrinter Agent. The latter agent is a non-primitive micro-agent which consists of two primitive micro-agents, one of which is called a DataCollector Agent - providing the data to be printed - and the Dumper Agent being able to process the actual printing. This scenario reflects all types of micro-agents and the idea of composition in an agent-oriented way, especially respecting the separation of concerns as the User Agent does not have knowledge about the primitive micro-agents within its interaction partner. Figure 3.2 visualizes this scenario reflecting the implementation done by the authors of the original concept. The numbers in the figure represent the sequence of execution.

However, the efficient means of communication have their price. The current implementation is simple to implement in first instance but lacks transparency at runtime. While a conversation manager is available for the agents micro-agent
communication is intransparent and cannot be retraced as neither conversational state nor past conversations as such are tracked. This raises the inherent problem of debugging as especially in the context of agents (of any size) information on their conversations are crucial to analyze system behaviour during development as well as runtime. Especially considering the potentially complex configurations when using agents on multiple levels developers demand for support to gain insight into the whole system.

Another limitation of the original implementation is the binding of receiver of messages by the according agents themselves. They can employ discovery mechanisms for lookup, the actual selection however is left to them. Here an alternative option to automatically bind a target agent based on system-wide selection strategies would be a convenient improvement. Given this, the strategy for all micro-agents could be changed platform-wide. This seems especially useful to optimize system performance in cases where this scope is beyond the individual agent and his developer. However, this should not override the preference of an individual micro-agent to select his conversation partner manually to keep a maximum of the already conceptually constrained autonomy.

Demanded by the authors of the original concept advantages using parallelism should be achieved by the system. This is only partially reflected in the original implementation of the micro-agents; the developer needs to take care of this
manually. While this characteristic might not have been so pressing earlier, in the context of Multi-Core Processing machines this demand becomes critical for overall system performance.

While the actual list of suggestions by the authors[71] exceeds this treatment one last idea to be mentioned is the support for atomic transaction by micro-agents in a fashion similar to the one in Database Management Systems (DBMS).

The key research problems to be addressed by this work are thus the lack of transparency in communication, the hard-coding of the actual agent binding in micro-agents and ensuring reliable runtime behaviour (avoiding of race conditions and deadlocks) while providing facilities for parallelization where appropriate.

As such the central hypothesis is to provide unified communication means for micro-agents to provide transparency for agent conversations, simplifying the development by support of dynamic binding as well as support of parallelization while maintaining a minimal performance loss. Given the original approach to fully implement the new elements based on Java a performance loss of 10% seemed acceptable. However, at this point it may be anticipated that the explorative nature of the approach taken led to significantly deviating results. The 'unified communication means' are decomposed into the following subtasks:

- Providing a message container for micro-agent communication,
- provide support for dynamic linking and binding of agents and
- reflecting the idea of parallelization in the concept.

As implicitly stated before, but elaborated at this point, any level of abstraction results in performance loss. As such micro-agents might probably best be implemented in Assembly. However, this would be an extreme beyond any compromise on a problem-focused agent-oriented view as introduced and differentiated in section 2.2. However, in contrast to the high-level FIPA-enabled agents fast interaction is the central feature of micro-agents which makes performance loss a sensitive topic when considering the additional abstraction introduced by the unified message container and dynamic binding. This characteristic of micro-agents
shall be treated as their efficiency principle. Further the principle of separation of concerns between the agents is central and loose coupling should kept as far as possible. A decision taken by the author to ensure the according comparability of performance is to keep changes to the actual micro kernel structure as minimal as possible.

At this point a clear conflict can already be perceived: While the separation of concerns supports the idea of unified communication both opposes the efficiency principle. This is the key decision to take whether to adopt or reject final results.

### 3.2 Design decisions

#### 3.2.1 Initial design

Given those objectives and requirements central observations as well as detailed design decisions will be retraced following this point.

The interaction based on direct method invocation allows two payload types which are goals and events. Goals are yet implemented from an empty interface with developer-defined attributes and methods. The goal thus encapsulates as much content as needed for the specific achievement. Potential results of goal execution are saved within the goals and passed back to the conversation initiator. Alternatively micro-agent roles can subscribe to events and be notified accordingly. Those payload types are the minimum to be supported by the message container and differentiated into the two according payload-related specializations. In the trade-off between expressiveness and efficiency the actual message container is modelled as a subset of the by far more complex FIPA message container available in the framework. This ensures coherence of concepts on the micro-agent level and supports the notion of 'ACL-like' interaction.

A further aspect for the whole design refers to the processing of the messages, firstly to support conversation tracking. To achieve this a central entity is suggested in order to complete message metadata (like conversation ID, message ID and the time of processing) in an unique and retraceable manner. Along with this
the according entity should provide the functionality of dynamic binding, conversation tracking and also log respectively store the messages in a history data structure.

Given those initial suggestions the current implementation for communicating micro-agents, thus the ones playing the "social role" (see Figure 3.1) were analyzed. As of the current implementation the according roles are differentiated by their communicative roles as either Provider and Customer. Dividing the according agents into these role ensures consistency when modelling the full system. However, a new layer of abstraction is necessary to hide the message container functionality. As such a new abstract `MicroMessageUser` role is considered whose sole purpose is to enforce an implementation of `send()` and `receive()`. Individual GoalProcessors and goal queues are attached to this `MicroMessageUser` in order to support asynchronous processing. This way the agent implementation needs to extend the `MicroMessageUser` and can thus implement send and receive according to their internal structure. Given the use of third party reasoning engines within an agent implementation only those two methods would demand for implementation in order to operate on the micro-agent platform. However, as the general approach to implement micro-agents will rely on the performative methods those are implemented by extending a `DefaultMicroMessageUser`. This implements the `send()` and `receive()` methods and maps those onto the performative methods of the developed agent and thus frees the developer from implementing send and receive. In consequence the role scheme of Provider and Customer is yet given up; the developer just has to deal with the `MicroMessageUser` role which should be able to participate in communication received from other agents independent from a 'Provider' or 'Customer' role.

Further implications for design amendments are derived from the review of the originally used benchmark code: Recalling the benchmark scenario and the according interaction scheme from Figure 3.2 it can be retraced that agents are loosely coupled by the use of the existing discovery mechanism - to be discussed at a later point - and invoked. However, this loose coupling is at least questionable as the `HelloPrinter` directly accesses the methods `collectData()` and
printData(). While this is one reason for the good result of the benchmark as it avoids four invocations per round the fulfillment of the principle of ‘separation of concerns’ is at least questionable. The original micro-agent concept however offers flexibility with regards to ‘responsive agents’ and their degree of loose coupling, especially when harming the efficiency property. For a generic communication including dynamic binding this degree of implicit knowledge about the internals of any agent on the platform playing ‘Collector’ and ‘Dumper’ role cannot be assumed. To reflect this a second variant of the benchmark was implemented. This does not assume any knowledge about its underlying agents and discovers them by their ability to execute the necessary goals. The interaction for this implementation is shown in Figure 3.3. The consequence could be anticipated: The execution of 10,000 ‘Hello World’ cycles for the second benchmark takes about 1.8 times (on average 300 ms) as long as the first one. Knowing that the result harms the efficiency requirement, it reflects a standardized interaction pattern which this work is about to establish on the micro-agent level. As such the result of the new benchmark is considered as the reference value of the ‘original’ framework and will

---

6The two direct invocations for the primitive micro-agents could be modelled by each an invocation on the want() method on request and the achieved() method on success as well as the internal invocation of the methods collectData() and printData() which would result in six as opposed to two method calls.

7Here again the numbers indicate the sequence of execution.

8Here again it is to state that standardization is to be treated as a means of abstraction and thus opposes performance.
serve as primary basis for comparison with the performance results after container integration.

Besides this ‘normalization’ activity further frequently occurring but unpredictable problems got apparent. Especially in the context of the frequently repeated use of the existing discovery and goal processing infrastructure concurrent access in the individual agent groups was observable. Reason for this is the ad-hoc fashion in which results for a discovery are collected by iterating through the capabilities (roles, goals) of each agent for a given group. This way only valid values are retrieved; even the agents rebuild their properties upon change to reliably avoid inconsistency. The developer of a system which makes heavy use of interaction might run into those concurrency issues again as the use of the performance-breaking lock-based synchronization of Java was avoided as far as possible. As such it is up to the individual developer to employ suitable thread safety measures to manage the trade-off of performance and reliability. Instead of addressing this issue directly in Java another approach was considered to circumvent this problem and keep those issues from the developer.

3.2.2 Clojure as Directory and Message Processor

Clojure, as a recent breed\(^9\) of high-level JVM languages - and earlier introduced in Section 2.4 - provides promising concepts which seem to match the issues identified and motivate its use to eliminate those. The relevant ones for this context are recalled at this place. One of those features is the management of mutable state by the use of STM instead of Java locking, the concept of agents and the focus on high performance as central design criterion. As Clojure directly operates on and inter-operates with Java - and putting the irony of using a functional language for the management of mutable state aside - it is qualified to be used as a means of prototypic extension for the current micro-agent framework.

As a consequence the design outline from above should be used with Clojure as

\(^9\)The stable version 1.0 was only released on 4th May 2009.
a basis to provide the agent directory and dynamic binding and as such directly invoke target agents of a message. However, when using Clojure to access Java code a new instance of Clojure is created when used in different classes which would only allow local execution. As most operation demand for shared data a central class should be responsible for handling all platform-related Clojure interaction. This function is assigned to the ClojureConnector which links the two layers. In consequence each directory-, agent-, role- or goal-related request is always backed by Clojure; the Java agents themselves do not contain data structures to represent their own properties. As such the ClojureConnector is modelled as Singleton\(^{10}\) which provides access to the backend in form of a numerous collection of methods directly invoking according Clojure functions. Its actual implementation is based on the Invoking Clojure functions from Java approach as outlined in section 2.4.3. Figure 3.4 shows an architectural scheme of the current implementation. For a more detailed perspective please refer to the class diagram of the amended micro-agent framework in the Appendix.

The Directory functionality is one of the central functions provided by Clojure. For the representation in Clojure a flat structure based on basic data structures has been chosen. Key focus for the choice was to optimize performance for query functionality while keeping the data structures simple for easy debugging via the REPL. The general associative data structure chosen is HashMap to ensure retrieval of results in constant time. A selection of central data structures is shown along with some explanation:

- **agentRole** maintains all agent-role associations. Key value is agent ID, multiple roles are saved as list in the according value.
- **roleAgent** is the counterpart for agentRole and is basically the inverted structure to resolve agents by role.

\(^{10}\)The Singleton choice in favour of static methods is grounded in the idea that a controlled number of ClojureConnectors could be of use at later point in time (e.g. multiple instances for testing respectively comparison at runtime); extension would then be easily feasible.
- **roleGoal** along with **goalRole** provide the equivalents to the above-mentioned for role and goals.

- **agentGroup** contains all agents belonging to a group of an identified agent and basically includes the full agent hierarchy.

The actual functions are majorly exposed for direct access via the ClojureConnector. As such agents can invoke the same methods as in the original framework to query agents, goals and so on as those are all mapped against the according ClojureConnector methods.

**Dynamic binding** in turnaround is fully realized by Clojure and as of now considered to be the default mode of execution. Given that all agents for a platform are registered on instantiation only a minimum of information is needed to initiate a conversation. Agents respectively MicroMessageUsers provide a performative to signal the state of the conversation and a goal instance. However, as this information is not sufficient to provide full conversation tracking the Goal interface of the original micro-agent framework was changed to an abstract class introducing the attribute goalID which represents a unique Goal Identifier. This enables
the tracking of a goal throughout whole conversations. Given this the message
container is instantiated on the sender side\(^{11}\) and according sender information is
added. The message container is then passed to Clojure. In Clojure first a check
on the performative is pursued. This underlies simple rules such as a conversation
ruleset which identifies the validity of performative sequences. Once passed the
recipient is determined based on the candidates being able to execute the goal.
Yet the standard strategy for finding an agent is to use the first one returned.
This agent is set as recipient in the message container and metadata like conversa-
tion ID, message ID and time of processing are added. Conversation partners are
saved along with the goalID and the last performative of the conversation. After
putting the message into the goal queue of the recipient it is stored to the history.
The flexibility becomes apparent once the original recipient replies. As the con-
versation is tracked per goalID, latest performative and conversation partner the
according recipient can be easily determined. Once the conversation tracking has
reached a final state like ACHIEVED or FAILED the conversation is considered
as finished and deleted from the set of active conversations.
This set of active conversations is transparently accessible at any time and thus al-
 lows tracking of all ongoing conversations. This can be made externable accessible
using the ClojureConnector. As such it could be integrated into administration
tools to monitor activity on the micro-agent level. Concluding this, the function-
ality of dynamic binding thus relies on three basic assumptions:

- Goals are identified by goalIDs which do not change during a conversation.
- Conversations only involve two agents.
- Conversations follow fixed performative sequences.

A full interaction sequence of a REQUEST and COMMIT is visualized as se-
quence diagram in Figure 3.5. Given the available Clojure functionality a further
approach targets towards **direct execution of Clojure code** in the framework.

\(^{11}\)To be precise: The message container is created in the send() method in the default imple-
mentation for MicroMessageUser.
To keep the consistency of the framework the Goal specialization ClojureGoal was established. Additional to goalID goals of this type provide a namespace, a binding of environment variables and an according Clojure command. When sending a request with this goal it is identified within the first execution step (namely in the MessageProcessor which now merely acts as preprocessor) and executed in an isolated Clojure instance using the Java interoperability approach *Running Clojure code from Java* (see section 2.4.3). The result is saved within the goal instance and passed back to the sender with the performative ACHIEVED. As such a simple execution of isolated Clojure code is feasible for individual agents. Direct execution of Clojure code in the ClojureConnector instance is not supported. Central reason for this is not to harm the consistency of its data and functionality\(^{12}\). From the agent point of view it does not make sense to get access to this low-level (and runtime critical) information. However, the introduction of a separate shared instance - solely for the purpose to interact via ClojureGoals - could be considered for further developments.

The state of this development is documented in the class diagrams provided in

---

\(^{12}\) Access would not only allow manipulation of data but also functions.
Appendix A. Further sequence diagrams clarifying directory and interaction functionality of the Clojure-backed micro-agent framework are located in Appendix B.
Chapter 4

Performance Optimizations and Results

Design and initial implementation were clarified in the preceding chapter. This chapter in turnaround focuses on the runtime results of the implementation, discusses the means of tuning and provides an overview of the final results.

4.1 Performance Optimization and Testing

The runtime performance of the initial implementation was beyond any acceptable value. Repeatedly achieved results lay around a value of 25 as compared to the base benchmark implementation (see Figure 3.3 to recall the reference benchmark scenario). After reviewing the activity flow throughout the system four main potential problem areas and according actions were taken into consideration to improve the results:

- Review of data structures
- Clojure-related performance optimizations
- Java interoperability
- Conceptual aspects

Reviewing the data structures was narrowed to a scope of functionality used to perform discovery and binding. An example for a Clojure function to retrieve the first agent being able to perform a given goal is:

\[
\text{(first \ (roleAgent \ (first \ (goalRole \ goal)))})
\]

This function resolves the first role being able to execute the given goal and returns the according agent playing this role. As of its simplicity it does not provide further improvement potential.

Continuing this process another structure respectively feature to be reviewed were the Clojure agents. The understanding of agency in this concept is limited to the weak notion as Clojure agents are reactive but do not provide any autonomy features. Clojure agents encapsulate some state and process any function sent to them on this state in an independent and asynchronous manner. As such an agent as well as a function which is to be sent need to be defined. The following example shows those elements and the according send command. An important detail on implementation is Clojure’s behaviour to automatically pass the calling function name as first parameter to the function. As such this needs to be anticipated in function design (see parameter ‘caller’ in the example).

;function to be sent to agent
(defn- \ run-msg-receive [caller message]
    "function for message dispatch by agent"
    (. (.getReceiver message) receive message)
)

;define agent
(def a (agent "Accessor 1"))

;send the actual function to agent
(send a run-msg-receive message)
However, functions are supposed to change the state of the encapsulated state. This functionality is not used at all. Agents in this example rather act as a replacement for a message queue and "abuse" the Clojure concept for Java access. Agents in Clojure run in Java thread pools. This keeps developer from any further thread management. The Clojure agents work reliable but an alternate Java implementation of message queues using the Producer-Consumer pattern was done which revealed that the use of Clojure agents was by about 20% slower than the Java queues. Reviewing the source code of the agent implementation and amending the thread pool size\(^1\) revealed no significant improvement. As a matter of fact the best performance with Clojure agents was achieved when reducing the number of threads to the number of processors. As a result of this the Clojure agent concept was not further used but the original notion of Java-based GoalProcessors was kept.\(^2\)

Further optimization potential is offered by Clojure itself. As it is directly works on Java it falls back to its data structures under the hood. Even though it provides dynamic typing those restrictions become apparent as Clojure produces errors when adding objects of different type to a structure which has been initialized using another distinct type. Access performance can be seen as major driver for this design decision. The same problem affected the initial implementation as well. The solution in Clojure is to do \textit{type-hinting}, thus indicate expected data types for method parameters. However, this should be focused on the cases in which Java reflection is needed to determine the type. It should not be considered as a general implementation principle. To find out the cases in which type-hinting can improve performance Clojure provides a *warn-on-reflection* flag which indicates the lines of code along with the according code piece demanding for Java reflection. This way all functions marked as such were provided with type-hints.\(^3\)

\(^1\)Rich Hickey chose 'number of processors + 2' as his magic formula for the thread pool size (see clojure.lang.Agent for implementation).

\(^2\)The design in the preceding section already reflects the latest state, firstly to avoid confusion about different designs, secondly because the relevance of the Clojure agents seemed limited in the design introduction.

\(^3\)Tests were run with Clojure 1.0 (stable) as well as the current Clojure 1.1 (alpha). Neither performance nor relevant functional differences with regards to the implementation were apparent.
Doing so in turnaround breaks the principle of dynamic typing. One major experienced benefit from dynamic typing on reconstruction of the logical backbone in Clojure was a implementation of a simplified agent framework focusing on the central entities (agents, roles, goals) to test the suitability of Clojure at all. This code could then iteratively be developed in this development framework. In this context dynamic typing was very helpful as agents, role and goals could be typed as strings; the developed Clojure code could then directly be used with the OPAL micro-agent framework without any code amendments. Type-hinting broke this useful support. As of the progress this did not turn out as a central issue. For testing purposes, however, the agent platform prototype had been very useful. An example for type-hinting is given below. In this case the parameter message is hinted as MicroMessage.

```clojure
(defn set-message-id [#^org.rakiura.micro.MicroMessage message]
  "sets the message id before sending the message"
  (.setMessageId message (count (messageHistory (.getGoalId (.getGoal message)))))
)
```

The effect by the modifications in functions using Java reflection was the strongest of all tuning measures taken and improved the benchmark speed by more than 400%.

A further tuning consideration, testing the effect of Java interoperability, should be undertaken. This refers to completing the implementation based on the original design without Clojure. The major functionality is provided and served as a reference for the design backed by Clojure. However, this was not completed yet as priority was put on improving the Clojure implementation rather than establishing the Java version.

Conceptual considerations, such as the use of a complete conversation ruleset containing a full sequence of 'REQUEST', 'COMMIT' and 'ACHIEVED' (in this example related to the positive goal achievement), provided further overhead compared to the slim original benchmark scenario. In favour of efficiency the demand to 'commit' was removed from the original conversation rule set. This decision is realistic as the communicative constraint confounded the comparability⁴. In the

⁴The original benchmark does not include any 'commits'. 

To find out further improvements the application runs were profiled using the Eclipse Test & Performance Tools Platform[86]. In the Memory analysis a frequent use of character arrays was apparent. Tracing back this use it was relating to Clojure’s native data structure implementations. As such a deeper analysis of the individual Clojure data structures could be a promising approach to achieve further improvements. Part of this investigations would again point to the alternative implementation in pure Java.

Summing up the tuning, apart from minor code improvements the optimization means outlined before resulted in the highest performance improvements. Other Clojure-specific suggestions like the use of data type primitives and the strict use of recursion were reviewed but mostly did not apply for the implemented functionality.

Considering testing and debugging two mechanisms of Clojure proved to be extremely helpful: The first of those is the already mentioned dynamic typing which helped to systematically narrow down error source in the implemented functions by using a simplified subset of the agent framework. Another valuable helper was the availability of the REPL access. For this context the REPL was bound to a TCP port and thus was accessible with any telnet client. In combination with a Java debugger respectively at runtime in general this proved very helpful as change could be easily retraced and additionally offered the potential for interaction with the platform at runtime.

### 4.2 Benchmark results

After reflecting the design, implementation and even tuning the reader still lacks some indication how the actual system performs in comparison. Along with this some of the earlier introduced platforms will additionally be taken into account. Table 4.1 lists the comparison between the latest state of all implementation specializations including results of the other tested platforms (see section 2.3.3) for
absolute and relative comparison.

<table>
<thead>
<tr>
<th>Agent platform</th>
<th>Runtime in ms</th>
<th>Rel. diff. to O</th>
<th>Rel. diff. to R</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPAL Micro-agents - original (O)</td>
<td>170</td>
<td>1</td>
<td>0.56</td>
</tr>
<tr>
<td>OPAL Micro-agents - reimplementati</td>
<td>302</td>
<td>1.8</td>
<td>1</td>
</tr>
<tr>
<td>OPAL Micro-agents - Clojure, with</td>
<td>1596</td>
<td>9.4</td>
<td>5.2</td>
</tr>
<tr>
<td>history</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPAL Micro-agents - Clojure, w/o</td>
<td>1279</td>
<td>7.5</td>
<td>4.2</td>
</tr>
<tr>
<td>history</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPAL Micro-agents - interpreted</td>
<td>5536</td>
<td>32.6</td>
<td>18.3</td>
</tr>
<tr>
<td>Clojure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MadKit</td>
<td>781</td>
<td>4.6</td>
<td>2.6</td>
</tr>
<tr>
<td>JADE</td>
<td>6781</td>
<td>39.9</td>
<td>22.5</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of micro-agent implementations and further platforms (10000 'Hello World' iterations)

The figures presented demand for a little clarification. All test were run on multiple machines\(^5\) and confounding factors like caching were eliminated as far as possible by reinitializing Java for each run. The average value of 20 rounds of the according benchmark was taken as performance value. The values represent the results for the faster CPU.

The by far fastest implemented micro-agent variant was the original one (in the table abbreviated as O) implemented by the authors of the original micro-agent framework (to recall the scenario refer to figure 3.2). The comparison to a native Hello World implementation was omitted as the statistical difference did not allow to conclude any real performance difference. Major reason for this is presumably the fact that both implementations’ performance is only constrained by Java I/O performance - given today’s CPU speed. In order to identify differences the number of HelloWorld iterations should have been increased to a by far higher value. However, for this context it shall be enough to consider that the original micro-agent implementation is equal to native implementation.

\(^5\)Test machines: Intel Core2Duo 2.4Ghz, 3GB RAM, Microsoft Windows XP Professional, JVM 1.6.0_12; Intel Core2Duo running at 1.33Ghz, 2GB RAM, Microsoft Windows XP Professional, JVM 1.6.0_16.
The reimplemented Micro-agent version (abbreviated R) provides a "fair" means of comparison as of the strictly performative-based interaction (see figure 3.3 for scenario) thus serves as a baseline for the "original" micro-agent framework. The actual Clojure-based benchmark, making use of dynamic discovery and binding including the history, needs 5.2 times as long as the baseline of this comparison. The history as of current state is a simple list of all transferred messages. As it is not further processed - focus of optimization was on the actual Clojure message routing and dispatch - it yet leads to a memory overflow when using higher values than 50,000. To provide a "cleaner" functional comparison the history was excluded for an additional run. Thus the Clojure-backed micro-agent framework without history achieves a result of 4.2 times the runtime of the baseline and is about 7.5 times slower than a native Java calls.

These findings show a significant difference to the slim original micro-agent framework. Here the trade-off of usability and performance must be considered. Using Clojure provides the reduction of any conversation to a simple send() method including goal to be achieved and performative expressing the sender’s intention. As Clojure is focused on high performance better values could have been expected. Besides any further optimization approaches the interfacing between Clojure and Java might be another bottleneck - or at least an unnecessary deviation - of the full interaction. A full implementation of the agent infrastructure in Clojure and the according instantiation of agents from Clojure could be an answer. Those could then be modelled to consistently comply with the functional programming paradigm. But considering this in consequence should not harm the agent-oriented development inside the then "functional container".

Running Clojure in interpreted mode turns out to be very slow. Reason for this is the fact that the full procedure of binding a namespace, the according variable bindings and the actual compilation of the Clojure code need to be done every round. This processing and memory intensive task could be replaced by considering a local Clojure instance for each agent. Execution would then not enforce reinstantiation. Again, if the whole framework was implemented in Clojure this

---

6This 'intention' shall not be interpreted as the concept from the BDI model but as an illocutionary act.
would not be worth arguing as it then would be the 'native' way of processing for agents anyway.

Given those results and comparing those with the benchmark results for MadKit and JADE it is clearly visible that MadKit is certainly faster than the Clojure-backed micro-agents. Nevertheless, when taking the results of JADE into account it gets apparent that the Clojure-backed micro-agents belong to the cluster of the 'fast' agents - despite a doubled execution time compared to MadKit.

This gets even clearer when reviewing the runtime behaviour of the agent implementations. In order to identify performance leaks and making predictions about scalability of the implementation the number of benchmark rounds was increased up to a number of 100,000 rounds. Figure 4.1 shows the absolute results based on runs for values 10,000, 20,000, 30,000, 50,000 and 100,000.

From this figure the 'clusters' of agents get very obvious. Only for the heavy-weight FIPA-based JADE implementation run time grows exponentially. The JADE implementation did not process the benchmark beyond 40,000 iterations and eventually ran out of memory. To get a better view of the detailed behaviour

\[ 	ext{Figure 4.1: Absolute runtimes for values up to 100,000} \]

\[ 7 \text{This refers to the analysis of the initial comparison done in section 2.3.3.} \]
figure 4.2 concentrates on the 'fast' agents. Analyzing this graph, it is apparent that the runtimes of the fast agents are fairly linear; MadKit takes about 2.5 times as long as the reimplementation of the original benchmark, the Clojure-backed micro-agents take a little less than double the time of MadKit. The Clojure-backed micro-agents with history ran out of memory beyond 50,000 iterations. The reason as outlined lies in the incomplete processing; the flexibility of Clojure data structures makes it tempting to keep the history in memory. This issue can be easily solved but priority was put on achieving an acceptable overall performance.

The relative changes in performance for up to 100,000 iterations (compared to the results for 10,000 iterations) offer a more detailed view and raise potential for scalability predictions. Figure 4.3 represents the complete data set which again shows the different performance clusters. Looking at the detail view (see figure 4.4) offers interesting findings. Even though all agents showed positive change of relative performance the original - close to native - implementation experienced a steep decline after its initially high values. The results were achieved repeatedly but no relation to the garbage collection of Java could not be found. More

![Figure 4.2: Detail view on absolute runtimes for values up to 100,000](image)
interesting than this is the performance development of MadKit as the relative performance was steadily increasing resulting in a speed up of 20% on 100,000 iterations. To reveal the reason for this a deeper analysis of the current MadKit seems indicated for further research.

Summing up the results of the running system it is to say that the expected increase of maximum 10% could not be held when attempting the implementation of the message container in Clojure as results lie - depending on message container version - around 420 to 520% of the already reimplemented agent benchmark. As such the original hypothesis outlined before doubtlessly failed but did not anticipate this explorative deviation from the pure implementation based on Java - which as of focus on Clojure has not taken place.

Keeping this in mind the approach was a failure on the 'hard criteria' side. However, the use of Clojure offered new capabilities which in turnaround had not been anticipated when setting up the hypothesis and thus should be treated separately as to be discussed in the Outlook for this research.

**Figure 4.3**: Relative performance changes for up to 100,000 iterations
4.3 Open Issues of the work

Before proceeding to the discussion of the system’s potential and continuing the ongoing hunt for performance some issues/limitations of the current implementation still need to be outlined. Most of those are simple to establish but demand for thorough consideration, especially when considering the performance focus of the whole system:

- The history cannot be held in memory but needs to be written to disk. Fast approaches to do so were tested (e.g. sending the history to a Clojure agent to deal with this matter, handing the history management back to native Java and handling in a separate thread) but not fully implemented. Performance tests for different approaches need to be undertaken (yet it is clear that handling it within Clojure promises the best results). However this issue does not restrict the tracking of active conversations (see section 3.2.2).
• The event subscription system of the micro-agent framework is not yet "translated" to Clojure; only the goal-based interaction is fully functional.

• Conversation tracking functionality is available but not connected to any kind of administrative tool to make use of it.

• To provide a better performance comparison and isolate the dynamic binding functionality of Clojure the discovery ability - without binding - should be provided to only leave the agent directory functionality to Clojure but actual binding of the recipient to the individual Java-based agent.
Chapter 5

Conclusion and Outlook

After providing details on the work undertaken this final chapter sums up the structure and findings of the work and provides an outlook on promising directions of research.

5.1 Conclusion

The work undertaken provides a structural amendment of the existing micro-agent framework. In preparation to this a literature review of agent-related concepts like architectures, means of knowledge-level communication and Agent-Oriented Software Engineering along with its potentials and problems is given (section 2.1). Following this agent platforms as a base for implementation were mentioned. As such the probably best-known standardized agent architecture specification was introduced. In order to get a better understanding of alternative agent concepts in the context of implementation different agent platforms were compared based on their agent concept, communication means, ability to solve more complex reasoning tasks and performance (section 2.3). The comparison isolated two clusters of agent platforms and motivated the focus on the Otago Agent Platform as of its comprehensive understanding of agency.
In a next step state-of-the-art developments from the field of the reviving functional programming were highlighted outlining their advantages over the use of native Java and their approaches to handle concurrent processing (section 2.4). Along with this the full Java interoperation potential for the later one, Clojure, was comprehensively compiled.

Given this background, the concept of micro-agents, though already mentioned during the platform comparison, was explicated on a more detailed level. Its yet unimplemented features as well as problems of the currently existing implementation were elaborated. Focusing on a set of issues and the according hypothesis the design and partially implementation decisions for a message container as well as the dynamic binding infrastructure were outlined (chapter 3). Given the chosen iterative approach and the increasingly explorative nature of the work the design and implementation phases, especially in the later process of development, showed strong interrelations.

Chapter 4 documented the process of performance optimization of the Clojure-backed micro-agent platform. Key improvements derived from the decision to abandon the Clojure construct of agents and the use of language-specific optimization mechanisms. Following this the final performance results of the implementation were put into the context of the earlier introduced benchmark. The trade-off of increased loose-coupling, convenience for the developer and performance was discussed in the context of the results. Considering the aforementioned hypothesis the implementation did not meet the requirements of minimal performance loss but offered new potentials which had not been anticipated at an earlier point but shall be discussed to provide further suggestions for research in this area.

5.2 Outlook

Apart from the not fully completed representation of the concepts given in the original framework (see section 4.3) the current result can serve as a base for further elaboration of the micro-agents beyond the scope of the original concept.
In the following a few of those considerations - partially simple, partially more far-reaching - will be discussed:

- The current improvement was focused on the inside of the agents, easing the developers’ lives by relieving him from thread management and agent coupling issues and providing monitoring capabilities. However, the effect of the micro-agents is not sensible from the outside. *Cross-platform interaction* on micro-agent level is the next issue to tackle to realize the performance advantage of micro-agents beyond an isolated runtime environment. But the rising complexity of the multi-level communication needs to be considered carefully; both high-level agents and micro-agents can then communicate autonomously. Effects on the relation/coordination between the two different levels needs to be investigated. Firstly conversations on different levels need to stay isolated, secondly tools are necessary to ensure that the agents’ ”minds” are transparently retraceable (in terms of debugging). The potential however could be ”to switch” between means of communication (high-level versus low-level communication) depending on bandwidth restrictions (especially in the context of mobile devices) and knowledge about the target entities (capabilities of target agent platform and receiving agent).

- The use of *Clojure code as communication payload* between agents is another step to take. One motivator is the compact representation, another the potential to make use of the functional capabilities to use the payload as direct input for the receiver’s agent function(ality). Referring to FIPA’s content language syntax as being ’LISP-like’[22] Clojure seems to be a perfect match as content language for fast agent communication.\(^1\) Given the efficient use of this it could even result in performance improvements if consistently relying on the functional representation of message payloads and - as a further consequence - agents themselves.

\(^1\)This idea could be extended from the micro-agent arena to also introduce a Clojure syntax-based content language for the heavier FIPA message containers.
• *Agents developed in Clojure* could be fully functional oriented and/or construct "their" Java container to satisfy both the functional and object-oriented development languages. This development would imply that the answer of the question about a prevalent base language for the implementation of the agent framework (Java versus Clojure), brought up in section 4.1, targets towards Clojure.

• Given the interaction between (then homogeneous) platforms the problem of *agent mobility* could be tackled. Agents (as Clojure code) could be transferred to another platform, may it be to improve performance based on locality of execution or redundancy.

• Similar to the suitability of Clojure to be used as a base for a 'content language' the nested S-expression structure could be a 'natural way' to express goal (de)composition (in a sense of *feeding goals with goals*). As such this provides the opportunity to supply micro-agents with a light-weight approach of implicit reasoning about goals (by allowing declarative tracking of goal decomposition and execution on 'owned' micro-agents) as demanded by Braubach and Winikoff (refer to section 2.1.3 for full context).

• Given Clojure as platform base, *Interactive development of and real-time interaction with agents* is only a step away. Even as of now the only task would be to wrap a REPL into an agent container and invoke action via a (yet) telnet user interface (as outlined earlier). From this even Java code could be used in an interactive way. This would lever the usability of the agent platform to a different level and - given a toolset integration in the according user interface - contribute to new style of developing agent-based systems.

• A missing element to test all the functionality and to also put the developed software into a reasonable context with more realistic performance measures the need of *Use Cases* is clearly indicated. Use cases should reflect multiple agents with overlapping capabilities, more complex tasks demanding for a realistic degree of micro-agent decomposition and making use of goal-
well as event-related interaction. Having those use cases would not only support a reliable assertion on performance and suitability of micro-agents (e.g. indicate up to which degree simple interaction mechanisms on micro-agent level are suitable or the use of ACLs with stronger expressiveness is more useful). This could drive implementation-related decisions or amendments of the current design which is yet based on the simple benchmark scenario.

- Last but not least the vision to unify the multi-level agent concept of OPAL under a common *Graphical User Interface* should be kept in mind. The unified communication is an important step towards a plug-and-play-enabled composition of a multi-agent system allowing the direct insertion of functional code for execution. Here the generation of Java code from Clojure would ensure reliable code generation and allow amendments of agents via the GUI at runtime. This functionality would not be available when considering a purely Java-based implementation. The unified approach of the mentioned elements would enable GUI-based interactive Agent-Oriented Software Engineering, thus supporting fine-grained design, implementation and maintenance at runtime\(^2\) - a unique feature integrating Clojure, AOSE and the micro-agent concept.

Summing up a gros of the issues mentioned it gets apparent that functional decomposition (e.g. based on Clojure) might be a more genuine approach to model multi-agent systems than the structural decomposition based on object-oriented principles following a classical imperative programming style. At least for the typically fairly simple 'collective' micro-agents - in contrast to the 'individualistic' high-level agents - this could be a promising approach. Spinning this vision further, actual agent entities demanding for stronger reasoning capabilities could employ reasoning using logic programming (e.g. AgentSpeak) which would bring those orthogonal paradigms together in the context of OPAL\(^3\). However, for now it is to decide if Clojure’s performance is sufficient to act as a base of the micro-agent

\(^2\)The interactive approach implicitly unifies development and runtime anyway.

\(^3\)Though this approach of unification is different from - and presumably by far less efficient than - the ones sought by numerous researchers as provided in \([87]\) and \([88]\).
framework. From the list of findings and potentials the author suggests to pursue this effort and solve this trade-off in favour of Clojure’s agent-friendly powerful interactive features. Given this case numerous improvements of the current framework can be envisioned - bringing expressive agent capabilities down to a fast and comparatively efficient level of execution. The holistic understanding of agency in OPAL makes it unique amongst other frameworks and, given further extensions on its lower level as well as integration with AOSE, could narrow the semantic gap between both agent levels - getting closer to remove the yet exclusive ‘or’ in the question: Should multi-agent frameworks be fast or intelligent?
Appendix A

Class diagrams

The class diagram representing the current state of implementation is split into several figures. To differentiate between unchanged, amended and newly introduced diagram elements table A.1 documents this with reference to the according figure:
<table>
<thead>
<tr>
<th>Diagram element</th>
<th>unchanged</th>
<th>amended</th>
<th>new</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemAgentLoader</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.1</td>
</tr>
<tr>
<td>AgentController</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.1</td>
</tr>
<tr>
<td>AgentLoader</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.1</td>
</tr>
<tr>
<td>AgentState</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.1</td>
</tr>
<tr>
<td>InvalidPerformativeException</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.1</td>
</tr>
<tr>
<td>Performative</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.1</td>
</tr>
<tr>
<td>AgentController</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.1</td>
</tr>
<tr>
<td>ClojureMessageProcessor</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.1</td>
</tr>
<tr>
<td>ClojureScriptExecutor</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.1</td>
</tr>
<tr>
<td>ClojureConnector</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.2</td>
</tr>
<tr>
<td>Role</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.3</td>
</tr>
<tr>
<td>Group</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.3</td>
</tr>
<tr>
<td>GoalProcessor</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.3</td>
</tr>
<tr>
<td>AnonymousRole</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.3</td>
</tr>
<tr>
<td>DefaultGroup</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.3</td>
</tr>
<tr>
<td>Agent</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.3</td>
</tr>
<tr>
<td>AbstractAgent</td>
<td></td>
<td>x</td>
<td></td>
<td>Figure A.3</td>
</tr>
<tr>
<td>AnonymousAgent</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.3</td>
</tr>
<tr>
<td>SystemOwner</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.3</td>
</tr>
<tr>
<td>ClojureMicroMessageUser</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.3</td>
</tr>
<tr>
<td>DefaultClojureMicroMessageUser</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.3</td>
</tr>
<tr>
<td>MicroMessage</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.4</td>
</tr>
<tr>
<td>EventMicroMessageUser</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.4</td>
</tr>
<tr>
<td>GoalMicroMessageUser</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.4</td>
</tr>
<tr>
<td>ClojureGoalProcessor</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.5</td>
</tr>
<tr>
<td>DefaultClojureGoalProcessor</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.5</td>
</tr>
<tr>
<td>Goal</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.5</td>
</tr>
<tr>
<td>ClojureGoal</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.5</td>
</tr>
<tr>
<td>Event</td>
<td></td>
<td></td>
<td>x</td>
<td>Figure A.5</td>
</tr>
</tbody>
</table>

Table A.1: Table of changed elements in Micro-agent framework
Figure A.1: Class diagram of Clojure-backed Micro-agent framework 1/5
### Class Diagrams

**Figure A.2:** Class diagram of Clojure-backed Micro-agent framework 2/5

```plaintext
 ClojureConnector
 namespace: String
define int
 defAgent, Var
defSubAgent, Var
countAgents, Var
 isRegisteredAgent, Var
 addAgent, Var
 remSubAgent, Var
 isGroupAgent, Var
 isSubAgent, Var
 getGroupAgent, Var
defSubAgents, Var
defRole, Var
defRole, Var
 findAgentOfRole, Var
 findAllRolesForAgent, Var
 findRolesForClass, Var
 addGoal, Var
 delGoal, Var
defAllGoals, Var
defGoalsForAgent, Var
 findAgentPerGoal, Var
 findAgentPerGoalInGroup, Var
 executeGoal, Var
 sendMacroMessage, Var
 addMNUToAgent, Var
 remMMFromAgent, Var
 defStyleAttr, Var

class ClojureConnector()
definition ClojureConnector
 debugCut: int, message: String : void
 isAgent(agent: Agent) : Void
 registerAgent(agent: Agent) : void
 unregisterAgent(agent: Agent) : void
 isRegistered(agent: Agent) : Boolean
 addSubAgent(subagent: Agent, groupagent: Agent) : void
 remSubAgent(subagent: Agent) : void
 isGroupAgent(agent: Agent) : Boolean
 getSubAgents(agent: Agent) : Agent[]
 isSubAgent(agent: Agent) : Boolean
 getGroupAgent(agent: Agent) : Agent
 addRole(agent: Agent, role: Role) : void
 delRole(role: Role) : void
 delAllRolesForAgent(agent: Agent, role: Role)
 findRolesForClass(agent: Agent, class: Class) : Role[]
 findRolesForClass(agent: Agent, role: Role) : void
 addGoal(role, goal: Class) : void
 deleteGoal(role, goal: Class) : void
 deleteAllGoals(role, Role) : Void
 findRoles(role: Role, String: String)
 isApplicableGoalForAgent(agent: Agent, goal: Goal) : Boolean
 findApplicableGoalsForAgent(agent: Agent, goal: Goal) : Goal[]
 findAgentForGoal(goal: Goal) : Agent
 findAgentForGoal(goal: Goal, Agent)
 findAgentForGoal(agent: Agent, goal: Goal)
 findAgentsForGoal(goal: Goal, Agent)
 countAgents() : Object
 getAgentForRole(anonymousRole: AnonymousRole) : Agent
 sendMacroMessage(message: MicroMessage)
 addMacroMessage(agent: Agent, goal: Goal) : void
 remMacroMessage(agent: Agent, goal: Goal) : void
 sendMacroMessage(agent: Agent, role: Role)
 remMacroMessage(agent: Agent, role: Role)
 getActiveConversations() : Object
```
Figure A.3: Class diagram of Clojure-backed Micro-agent framework 3/5
Appendix A. Class Diagrams

Figure A.4: Class diagram of Clojure-backed Micro-agent framework 4/5

Figure A.5: Class diagram of Clojure-backed Micro-agent framework 5/5
Appendix B

Sequence diagrams

Sequence diagrams of the system interaction utilizing Clojure can be found at this place. The diagrams represent any exemplified message processing sequence in figure B.1 (already provided in figure 3.5 as small scale version) and directory interaction in figure B.2.
Appendix B. **Sequence Diagrams**

**Figure B.1:** Sequence Diagram of Dynamic Discovery and Binding
Figure B.2: Sequence Diagram of Directory functionality
Bibliography


