A Unified Approach to the Development and Usage of Mobile Agents

D i s s e r t a t i o n

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik der
Friedrich-Schiller-Universität Jena

von
Diplom-Informatiker Steffen Kern
geboren am 27. April 1979 in Erfurt

07. Februar 2012

Gutachter

1. Prof. Dr. Wilheln R. Rossak, Friedrich-Schiller-Universität Jena
2. Prof. Dr. Christian Erfurth, Fachhochschule Jena
3. Prof. Dr. Ilka Philippow, Technische Universität Ilmenau

Tag der öffentlichen Verteidigung: ....

Abstract

Mobile agents are an interesting approach to the development of distributed systems. By moving freely accross the network, they allow for the distribution of computation as well as gathering and filtering of information in an autonomous way. Over the last decade, the agent research community has decidedly achieved tremendous results. However, the community was not able to provide easy to use toolkits to make this paradigm available to a broader audience.

By embracing simplicity during the creation of a formal model and a reference implementation to create and execute instances of that model, our aim is to enable a wide audience – even non-experts – to create, adapt and use mobile agents. The proposed model allows for the creation of agents by combining atomic, self-contained building blocks and we provide an approachable, easy to use graphical editor for the creation of model instances. In two evaluations, we could reinforce our believes that, with the achieved results, we could reach our aims.

Zusammenfassung

Mobile Agenten sind ein interessanter Ansatz für die Entwicklung von verteilten Systemen. Mit ihrer Fähigkeit, sich autonom und frei im Netzwerk bewegen zu können, eröffnen sie neue Möglichkeiten für die Verteilung von Rechenzeit sowie für das Sammeln und Filtern von Informationen. In den vergangene zehn Jahren hat die Agent Community enorme Fortschritte und Ergebnisse erzielt. Allerdings ist es ihr bisher nicht gelungen, einfach zu erlernende und benutzerfreundliche Frameworks und Toolkits zur Verfügung zu stellen, welche einem breiten Publikum den Zugang zu diesem faszinierenden Paradigma ermöglichen.

Mit der Fokussierung auf Einfachheit und Zugänglichkeit wird in dieser Arbeit ein formales Modells zur Beschreibung mobiler Agenten und eine Referenzimplementierung zur Ausführung von Modellinstanzen erstellt. Dabei ist es das erklärte Ziel, einer größtmögliche Anzahl von Nutzern das Erstellen, Anpassen und Ausführen von Agenten zu ermöglichen. Dies schließt Personen, welche keine Experten auf dem Gebiet der Agentenforschung beziehungsweise Computerwissenschaften sind, explizit ein. Das entwickelte Modell erlaubt die Erstellung von Agenten durch die Kombination von atomaren, wiederverwendbaren Bausteine. Als Teil der Referenzimplementierung bietet ein grafischer Editor die Möglichkeit, Modelinstanzen für die anschließende Ausführung zu erstellen. In zwei Evaluationen konnten wir zeigen, dass die Ergebnisse dieser Arbeit einen wichtigen Schritt für die Erreichung der gesteckten Ziele darstellen.

Acknowledgments

First of all, I would like to thank my adviser, Prof. Dr. Wilhelm Rossak, for providing the opportunity to start this thesis as well as the ongoing support over the last couple of years. He granted me the greatest possible freedom to pursue the chosen topic and offered guidance and advice in times of need. I can't imagine any better conditions to conduct such a large project.

Thanks to Prof. Dr. Christian Erfurth and Prof. Dr. Ilka Philippow for being second and third reviewer of this thesis.

Over the years, several students have contributed to this thesis. I thank Reana Sommerkorn and Johannes Meißner for creating the graphical notation for TAMo and, again, Johannes Meißner for building the first version of the graphical editor. Further, I thank Sebastian Kuhs and Matthias Keil for all their efforts during the expert evaluation.

For their participation in the non-expert evaluation, I would like to thank Mareike Mähler, Yeliz Yildirim-Krannig, Fabian Wucholt, Alexander Birnkammerer and Conrad Wrobel.

Last but not least, I thank my parents for their unconditional support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation for this Thesis

Over the past two decades, the area of agent-based software systems has evolved from niche research area into a mainstream paradigm for designing and developing software. Alongside, agent research has spawned numerous subareas like agent communication and coordination, multi-agent planning and interaction, negotiation protocols, agent models and methodologies or mobile agents. Depending on the focus of those areas, the definition of an agent as well as their aims differ slightly. Whereas agents in a multi-agent environment are considered to solve complex tasks by coordinating joint efforts of all available agents, other types of agents pursuit goals on their own and are considered as personal assistants of their respective owners. Especially mobile agents, e.g. agents that are capable to move between different execution platforms, aim at providing asynchronous, unattended services for their owners.

Over the last decade, the department of software engineering at the Friedrich Schiller University has been heavily engaged in mobile agent research thereby creating several iterations of the mobile agent toolkit Tracy. Tracy is a highly modular system based on a micro kernel architecture which is responsible for basic system operations such as agent execution and scheduling. For any additional functionality like agent migration, message exchange or security measurements, Tracy features so-called plugins. It served as playground for conducting research on the migration process of mobile agents and aspects of interoperability between different agent toolkits. Moreover, it was utilized in teaching to familiarize students with this paradigm and inspire them to engage in research.

With regard to interoperability the agent model of the Tracy agent toolkit has been extremely lightweight and did not provide any basic functionality like life cycle management to build upon. Therefore, creating new agents meant to start from scratch or extract helpful parts from an existing agent. Whereas the first option imposes a significant burden and a lot of tedious, repetitive work, the latter one usually leads to subpar software quality. Furthermore, without any framework to depend on, the learning curve was very steep and students who wanted to learn agent development struggled heavily.

Over the years, many other agent toolkits have established frameworks and tools to create various kinds of agents. However, most agent models as well as frameworks require specific domain knowledge and familiarity with partly exotic languages. Such expertise is not very widespread among software engineers thus imposing a huge entry hurdle into agent-based systems and development.

1.2 Contribution of this Thesis

Considering the efforts necessary to develop mobile agents with the Tracy agent toolkit, we aim at establishing a completely new way to create mobile agents by focusing on simplicity and ease of use.

One of the ultimate goals of the personal assistant research community is to enable non-expert users to configure and use agents as their representatives. So far, the community has not been able to propose a capable, usable solution to this goal. In our opinion, the most important aspect of such a solution is simplicity. By hiding nearly all technical aspects and provide easy to use tools that actively prevent users from making mistakes, we aim at achieving exactly this goal.

In unison with simplicity arises another aspect that is required to ensure a wider adoption of the mobile agent concept: a very flat learning curve. New users need to gain successes very fast to dig further into the provided frameworks and tools and, in the long run, get to know them exhaustively, stick with them and create astonishing results.

Looking at other agent systems, one reason that prevents agent-based software to gain wider acceptance can be attributed to the need to be familiar with partly exotic and uncommon languages and development environments. The proposed system does only rely on wide-spread and well-known programming languages, frameworks and environments to reach a very wide audience.

Despite the strong focus on simplicity and ease of use, agents created with the framework and tools presented in this thesis are by no means limited in their capabilities when compared to agents that have been created using traditional techniques. They are equally powerful and are able to use all services provided by an agent system as well as utilize third party services.

To achieve these goals, we propose a development process that involves two independent tasks where each one is carried out by a specific role. First, there are agent developers, that create atomic, reusable building blocks such as message exchange, web service access or agent migration. Second, there are agent designers which, by combining the available building blocks in the desired manner, create agents. With the proposed separation of agent developers and designers and the strict encapsulation of self-contained, single purpose actions, we can significantly increase the technical quality of agents and thus enable better reusability, adaptability and maintainability.

1.3 Outline of this Thesis

The following chapter gives a comprehensive overview on agent research, its core concepts and current research areas as well as limitations and drawbacks. Thereafter, in Chapter 3 we will outline our motivation in detail, evaluate current solutions and introduce an example scenario that will serve as a reference point in the remainder of this thesis. Chapter 4 introduces the TAMo model and its core properties and elements followed by an in-depth presentation of its implementation in Chapter 6. The aforementioned tools, that allow for the creation of TAMo based agents are discussed next and in Chapter 8 the results of our evaluations are presented. Chapter 9 and 10 close this work with a discussion of the results and an outline of future research.

Chapter 2

Scope, Technologies And Related Work

The proposed work intersects with several research areas and is related to numerous technologies and concepts. The upcoming chapter will outline these connections and describe everything that is involved to provide a comprehensive picture how all these parts fit together. We will start with a general discussion on the structure of the Internet and current techniques and trends concerning the development of distributed application. During the course of this chapter, we will further drill down to more specific areas which are closely related to the presented work. By the end of this chapter, the reader should have a sound understanding of the ecosystem in which this thesis is situated.

2.1 The Network

One cannot overestimate the changes that the Internet has brought to our society and culture. The ability to access nearly any information at any time and to communicate with the whole world in real time has not only changed our daily life but complete cultures and civilizations. During its existence, the Internet itself has changed several times; not so much in its underlying infrastructure which still suffers from former, primarily military goals [Clark, 1988] but more in the kind of provided services and their accessibility [Yoo, 2010]. Started as a rather small set of connected mainframes that were accessed by time-shared terminals, it is now a network of hundreds of thousand interconnected networks around the world. Over the last decade, the Internet has extended its capabilities. Formerly used only for communication and information access by human beings, it has evolved into a network of distributed applications and machine usable services [McIlraith et al., 2001]. In the following, we will look at software architectures of networked applications as well as the kinds of services that are utilized therein.

2.2 Software Architectures for Distributed Applications

In this section, we will outline some of the most common architectures for distributed applications. Starting with the traditional client server approach, we will follow the evolution of those architectures up to the modern cloud paradigm.

2.2.1 Client Server

The Client Server paradigm is nearly as old as the Internet itself. In the early day, numerous terminals accessed data that was stored respectively calculated on few mainframes. Despite the fact that this paradigm has several drawbacks, e.g. the server being a single point of failure or load balancing problems if the number of clients varies significantly, it is still the most widely applied approach for distributed applications. Modern approaches, which we will discuss in the remainder of this section, overcome those drawbacks, but in their core, most of them are just an adaptation of the client server architecture.

As mentioned above, in the client server architecture, a large number of more or less capable clients access resources and services provided by a much smaller number of servers as depicted in figure 2.1. Clients will issue requests and receive a response with the results which is afterwards processed by the client. Depending on the kind of client, these results may need a different amount of processing. In a classical web application, where a server delivers web pages, the client is just tasked with rendering and displaying those pages. However, modern web application have a much more powerful client part and the server merely delivers raw data which is afterwards processed by the client and integrated into a browser-based UI [Paulson, 2005; Garrett, 2005; Lawton, 2008; Marchetto et al., 2008]. The same applies for pure software clients, that access remote services to combine the data provided by several servers [Tsai et al., 2009]. Here, the amount of client computation is rather high, too.

2.2.2 Peer-To-Peer

The term *Peer-to-Peer* is well known outside of the computer engineering community as it does not only denote an architectural style but it is also the common synonym for file sharing networks that apply this architecture. As the name suggests, every participant

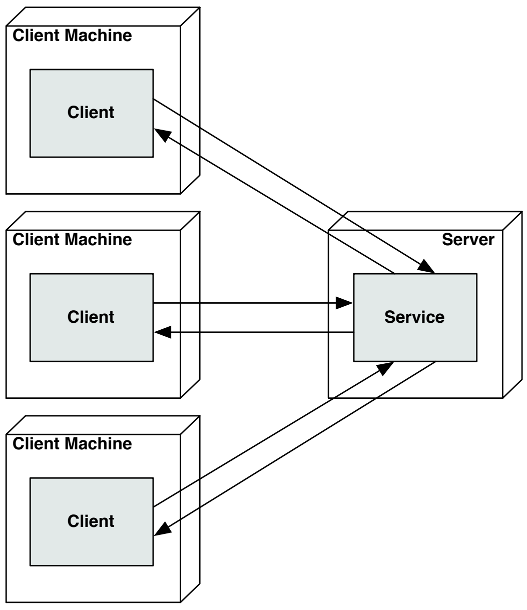


Figure 2.1: Client Server Paradigm

in such a network is an equal peer and data and information can be exchanged between any two peers. Thus, each peer can act as server and client as shown in figure 2.2. Joining a Peer-to-Peer network, a new peer must know at least a single other peer that is already part of the network. Some approaches introduce so-called *super peers* which never leave the system and act as entry point for new peers. Others use multicast like routines to find other peers.

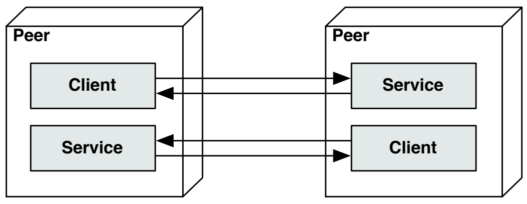


Figure 2.2: Peer-to-Peer Paradigm

Peer-to-Peer approaches overcome some well-known drawbacks of client server architectures [Vu et al., 2010]. These include, for example, scalability, reliability and performance issues in large scale applications. These drawbacks arise from the fact that the single server introduces a bottleneck and single point of failure. Although these problems can be overcome by spending more money on server hardware and infrastructure, a peer-to-peer approach seems to be a more elegant way.

Recent research on peer-to-peer architectures includes authentication and trust management [Moalla et al., 2010; Chu et al., 2010; Bachrach et al., 2009; Rodriguez-Perez et al., 2008; Mármol and Pérez, 2010] as well as tracking for so-called *freeriders* – peers that don't contribute to the system by only using resources without offering anything themselves [Tseng and Chen, 2011; Le Blond et al., 2009]. Due to the fact that data is spread across many peers, access to that data can be evenly distributed among all peers. This advantage is utilized for the distribution of large scale service offerings like Video-On-Demand [Lei et al., 2010b,a]. Peer-to-peer architectures are also widely used in Cloud services [Xu et al., 2009; Drost et al., 2011], which we will cover next.

2.2.3 Cloud Computing

Over the last few years, a new trend for providing services and data on the Internet evolved. It was started by large companies with huge data centers like Google or Amazon that wanted to leverage the immense storage and computing power provided by such centers. *Cloud Computing* [Armbrust et al., 2009; Hayes, 2008; Foster et al., 2008] aims at providing storage and computation in a transparent manner with a pay-per-use model. It allows for companies to extend their infrastructure very rapidly as the need arises without the burden to establish their own data center that would be capable to sustain peak load. This is even more interesting for small startups which can neither predict the evolution of their system's load nor can they finance setting up a data center. The cloud model can be compared to the production time frames in large scale fabs which allowed for smaller companies to develop microchips and move the production to a third party. A general overview on Cloud Computing can be found in [Chorafas, 2010].

The term Cloud Computing has been overused in recent years by labeling anything that somehow happens in the Internet as Cloud Service. Therefore, the term lacks a clear definition which led many to the question if „cloud computing is new wine or just a new bottle“ [Voas and Zhang, 2009]. But the core idea, receiving computation and storage power like electricity, is very appealing [Wang et al., 2010; Grossman, 2009; Kloch et al., 2011; Stanoevska-Slabeva et al., 2010].

Current research on cloud computing covers several important issues like the obvious security questions concerning one's own data hosted by a third party which is especially critical for a companie's intellectual properties or the personal data of any individual [Carlin and Curran, 2011; Yang et al., 2011]. Other researchers aim at providing an automated way to bring old legacy enterprise software into the cloud [Zhou et al., 2010a] or provide high performance computing similar to a Grid [Ekanayake and Fox, 2010]. Leveraging the power of the cloud in SOA architectures is another pursued topic [Shuang, 2010] [Rodríguez et al., 2010].

A topic, which is especially interesting to this work is the usage of mobile code as well as mobile agents to establish cloud services [Li et al., 2009; Aversa et al., 2010]. Due to the fact that computation and storage is extended in a transparent manner, services and data must be moved respectively distributed to other machines during runtime. Having a system that is able to transparently spread code and data in a cloud infrastructure is therefore a very appealing concept.

Concerning modern mobile devices like smartphones and tablets, the possibility to move computation from the device into the cloud is fascinating, too. Several years ago, this idea was spread due to low bandwidth and high cost of mobile data connections which made downloading huge amounts of data onto the devices unfeasible. Today, data connections are fast and cheap, however, the limiting property of current mobile devices is battery power. Thus, it is again desirable to move as much computational burden as possible into the cloud to preserve precious energy [Cuervo et al., 2010; Oberheide et al., 2008; Klein et al., 2010; Chetan et al.].

2.2.4 Mobile Code

Another architectural principle is *Mobile Code* [Ghezzi and Vigna, 1997; Carzaniga et al., 1997; Vigna, 1997]. It is not an alternative to the client server or peer-to-peer model as a whole but to the way of handling computation and data exchange between two communication partners. In the traditional model of accessing resources on a remote machine the requesting partner triggers the execution of a method at the remote machine or queries a remote database and receives the result data. In contrast, using the Mobile Code paradigm, code instead of data is transferred between the two communication partners to allow for a computation near the stored data. There are three different flavors of mobile code depending on the movement directions of code and data as presented in [Fuggetta et al., 1998]. In the *Code on Demand* model, software is send from a server to a client upon request to provide additional functionality at the client side. Probably the best know examples of this flavor are Java Applets and Ajax-based applications. The reverse model is called *Remote Evaluation*. Here, code is send from a client to the server to be executed near the data that is stored remotely. The main goal of this model is to avoid the transmission of large sets of data by moving the code to the data and not vice versa. Compare figure 2.3 and 2.4.

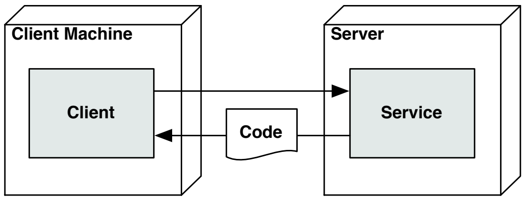


Figure 2.3: Code on Demand Paradigm

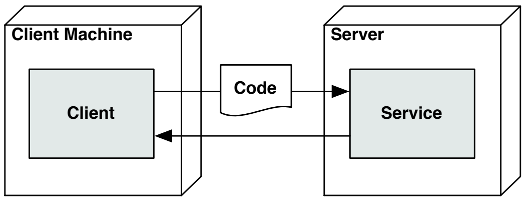


Figure 2.4: Remote Evaluation Paradigm

The third flavor of mobile code, *Mobile Agents* combine both aforementioned models and incorporate the autonomous agent metaphor. They are able to decide on their own, when and where to move and execute. The Mobile Agent paradigm will be discussed in detail on section 2.5. For a more comprehensive discussion on distributed systems we refer to [Coulouris et al., 2005].

2.3 Services

In this section, we cover so called web services. Such services transformed the Internet from a human usable network for information sharing into a network where the standardized interaction between machines and applications allowed for completely new kinds of distributed applications. A service can simply offer the current time of day but also it can also be a large scale meta-service that combines numerous other services. In this section, we will look at different service types as well as techniques to combine and utilize such services.

2.3.1 SOAP Web Services

So-called Web Services have been designed to allow for respectively ease machine-to-machine interaction in a heterogeneous network and are based on a number of open standards. A Web Service offers a certain functionality and has a machine-readable interface description usually provided in *Web Service Description Language*[1] (WSDL) format. Applications as well as other services can invoke a Web Service by sending messages as described in the services interface. Those messages are usually wrapped in a *Simple Object Access Protocol*[2] (SOAP) envelop and transferred using the *Hypertext Transfer Protocol*[3] (HTTP). To find appropriate services, one can look up a *Universal Description, Discovery and Integration*[4] (UDDI) service directory. Due to those open standards, Web Services allow for sharing of data and functionality in heterogeneous environments, e.g. across applications written in different languages and across operating system boundaries. Furthermore, services provided by different organizations can be composed into complex processes leading to *service-oriented architectures* (SOA). Discussion of these open standards is beyond the scope of this work so we refer to the literature [Erl, 2005; Krafzig et al., 2005].

2.3.2 RESTful Web Services

In his theses, Fielding [2000] proposed a new architecture for the description and usage of services based on the well-known and widely accepted HTTP protocol. This new architecture is termed *Representational State Transfer* (REST). In contrast to SOAP-based web services, which use only a small set of the HTTP protocol features, REST-based services fully leverage the power of HTTP thereby overcoming the need for a second protocol like SOAP.

With REST services, any entity that resides on a host is considered as a *resource* with a unique *URI* that can be used to access this resource. Using HTTP requests, the basic CRUD operations can be performed on any resource. The mapping of these operations to HTTP methods is straightforward with Create = POST, Read = GET, Update = PUT and Delete = DELETE. A Resource can be any kind of entity, from business model objects like customers or commodities to rather logical objects like the controller of a checkout process. To create a new customer, one would execute a POST

[1]http://www.w3.org/TR/wsdl
[2]http://www.w3.org/TR/soap
[3]http://www.w3.org/Protocols
[4]http://www.uddi.org/pubs/uddi_v3.htm

request on the URI of the customer container with the customer data in the request body. Or, to receive a list of all customers, a GET request could be performed on the same URI. REST distinguishes strictly between a resource and its probably various representations like XML, JSON, HTML or an Image. Clients can request a specific kind of representation by supplying an appropriate Accept HTTP Header alongside the request and the service is in charge to either deliver a corresponding result or reply with an error message.

HTTP is a stateless protocol and therefore REST services are stateless too. This leads to several interesting properties of this architecture. First, the server must not maintain any client states making it easy to scale the server side components. Second, clients are in charge to handle the application respectively session state. The proposed strategy to cope with this burden is *Hypermedia As The Engine Of Application State* (HATEOAS). Similar to links and references in an HTML document, the representation of a resource should contain references to actions, e.g. URIs, which a client can perform and which in the end will change the application state. For example, the representation of a shopping basket could provide a URI to start the checkout process as well as an URI to remove all items in the basket. This strategy provides two advantages: REST clients are not in charge to maintain their application state and the server still has control over the flow of actions without explicitly maintaining client sessions.

REST-based services were rapidly accepted by the industry and leading edge web application frameworks like *Django*[5] or *Ruby on Rails*[6] as well as state-of-the-art NoSQL databases like *CouchDB*[7] rely heavily on REST.

2.3.3 Service Composition And Workflows

Workflows have been introduced in the mid-eighties as a way to describe all aspects of office work [Bracchi and Pernici, 1984] including data, activities, employees and common processes. They have been derived from formal models like Petri Nets [Petri, 1962], production rules and flow charts [Wayne, 1973]. These models have been used to visualize and/or define processes. A workflow describes a business process including how tasks are structured, who is responsible for a specific task, what is the tasks execution order, the flow of information between tasks and how they are synchronized.

Later, development led to a closer relationship between modeling and the actual execution of workflows which resulted in complete Workflow Management Systems. With

[5]https://www.djangoproject.com
[6]http://rubyonrails.org/
[7]http://couchdb.apache.org/

such systems, one could define complex workflows that are afterwards executed by the system. For example, such a workflow could describe a billing process involving several activities like bill creation, inform the accounting department and send the bill to customer. For more information on workflows as means to describe business processes we refer to [Leymann and Roller, 2000].

As the name suggests, a workflow represents a flow of execution and is composed of several activities that are connected in their specific execution order. The flow of execution can be controlled by conditions and loops as well as forks and joins for parallel processing. Beside these, some workflow systems provide a way to handle exceptions and offer rules to bring the workflow back to an operational state.

Along with the evolution of web services, new workflow engines have been developed. These allow for the creation and execution of workflows, where activities are mapped to concrete web services. Thus, they provide a way to model business processes which are mapped to available services [Alonso and Mohan, 1997]. The current de-facto standard for describing processes based on Web Services is the *Business Process Execution Language 4 Web Services*[8] (BPEL4WS or short BPEL).

The drawback of such solutions is that the mapping between activities and Web Services has to be made by a developer in advance. Changes are tedious and error-prone and the execution of a workflow may fail if a single web service is unavailable.

A solution to this problem might be semantic service descriptions, which provide a way to map an abstract service request to a concrete web service dynamically. Thus it is possible to construct workflows without concrete service mappings but semantic descriptions of the required services. During execution, the workflow engine would try to find matching services for each activity. See figure 2.5.

With the ability to exchange concrete services, a workflow management system can adapt to changes during runtime. If a currently used service changes an important parameter like pricing or provided information, the workflow management system is able to find another equivalent service that operates in the anticipated way. As an example, consider a service offering airline travel information and a workflow for booking flights. The workflow relies on a booking service that delivers information like time schedules and available seats on a concrete flight. Now, the service changes and does only provide the number of available seats but not their position any more. As the workflow needs more specific information, the workflow management system will search for another travel information service, which still delivers the required information. Thus, with the

[8]http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

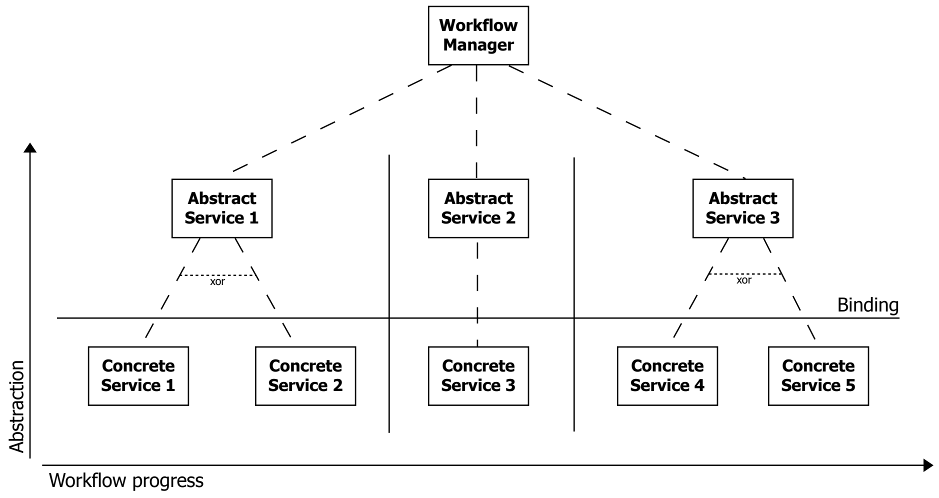


Figure 2.5: Structure of Workflow Systems. Binding of concrete services to abstract service descriptions is performed during runtime.

help of semantic descriptions and dynamic service binding, the system is able to hold and maybe increase Quality of Service (QoS) parameters.

2.4 Multi Agent Systems

The term *agent* has its origin in the Latin noun *agens* which means *the actor*. An agent is commonly understood as *a person or entity that acts on behalf someone else* or *a representative for somebody*. In some countries, the term agent denotes a person that works for the secret service. In english speaking countries, where the term has a more wider meaning, such an affiliation is better described by *operative* or *spy*. Agents are usually *domain experts* and can solve domain-specific tasks better and faster than others. For example, landlords can charge a real estate agent to find lodgers for their houses and take care of them during the tenancy. Another kind of agent are managers of artist or athletes who usually coordinate appointments and negotiate on new contracts. Another meaning of the term *agent* can be found in the area of chemistry and biology, where an agent is considered as a substance that initiates a process.

In computer science, software agents are – in general – programs that are able to solve a given task autonomously. Software agents have evolved from traditional object-oriented modeling and programming languages and the following *actor model*, that was introduced in 1973 by Hewitt et al.. The actor model introduced asynchronous messages and real concurrent execution – two properties which are associated with agent systems today.

Unfortunately, the agent community can still not agree on a single definition for software agents and a lot of attempts and formulations can be found in literature. While most definitions differ in certain aspects and details, nearly all agree on four main properties of agenthood that have been stated by Wooldridge and Jennings in 1995:

autonomy After given a task, agents act on their own without permanent control or intervention of their owner. They have control over their actions and their internal state. Thus, agents are well suited for asynchronous, non-time critical activities.

social ability Agents interact with each other using some kind of *agent communication language*. Moreover, they interact with their owner – usually by means of a graphical user interface.

reactivity Agents perceive their environment and will react on changes that occur in it. The environment may be a single agent platform, a network of several platforms or the whole Internet. The agent itself decides whether, when and how to react on such changes.

pro-activeness Despite their ability to react on changes in their environment, agents are able to actively decide what to do while pursuing their goals. Wooldridge and Jennings describe this with *taking the initiative*.

Several other attributes are often associated with software agents; however, they are not necessarily required for agenthood. These attributes include mobility [White, 1997], learning, and benevolence as well as rationality and veracity [Galliers, 1988]. We will take a closer look at mobility in section 2.5. Learning denotes the ability of an agent to acquire knowledge from its past experiences and executed actions. During its ongoing lifetime the agent will take into account that knowledge when considering its next steps. Learning should help agents become more efficient over time. Benevolence describes the fact that an agent does not have conflicting goals and will always try to fulfill its given tasks. A similar concept is meant by rationality, which indicates that an agent will work towards the achievement of its goals and will not forcefully try to prevent one or more goals from being achieved. That an agent will not knowingly distribute false information is called veracity. Huhns and Singh [1999] developed a test for agenthood that treats a tested program as a black box and only considers its behavior and not the internals. A more comprehensive discussion of agent attributes and the attempt to formalize them can be found in [Goodwin, 1993]. For further discussions on agents and agenthood, the reader may have a look at [Genesereth and Ketchpel, 1994; Luck and d'Inverno, 1995, 2001].

2.4.1 Agent Models and Architectures

Agents are situated in an environment and usually have some kind of sensors to perceive changes in that environment. Based on these inputs agents generate an output or perform actions trying to affect the environment and achieve their goals. Figure 2.6 depicts this simple concept. Over the years, many refinements have been developed including different kinds of sensors and actors to generate special input respectively output. Others have focused on techniques to transform inputs to concrete actions ranging from simple reactive models up to complex reasoning and planning. We will discuss several techniques to handle that input-output cycle in section 2.4.1.1 and 2.4.1.2.

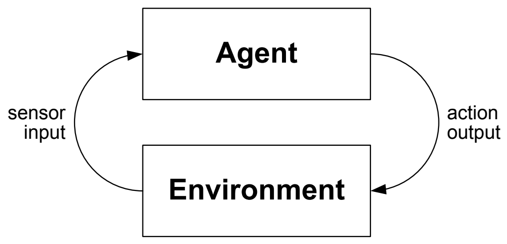


Figure 2.6: Agent and Environment Interaction. Adapted from [Wooldridge, 1999]

Possible environments may be a warehouse where softbots are ordered to organize the commodities, another planet reached by a space probe, or a number of computer systems in a network where software agents interact with each other. As for the used sensors and actors, one may imagine a softbot ordering and stacking crates and tons in a warehouse, who perceives its environment with visual, audio, and touch sensors while altering this environment with robot arms, a lifter or simply by moving around. Contrary, a software agent would perceive its environment by incoming messages and information provided by an agency. It could try to affect the environment by sending messages to other agents or users and use services provided by the agency. Regardless of the type of the environment, most of them will be huge, complex and heterogeneous, populated with agents, obstacles and different other systems. So it is unlikely that a single agent has complete knowledge of or control over its environment. At most, an agent will be able to influence its environment what leads to some interesting properties of agent systems. First, an agent will decide on its limited knowledge, which may be outdated or simply incorrect. This may lead to actions that may by no means be perfect in a given situation or produce the desired outcome. The agent should anticipate that the same action can produce different outcomes in similar situations or can actually fail. Thus, agents must cope with failures and uncertainties. In 1999, Wooldridge stated that – in most cases – *environments are non-deterministic.* Russell and Norvig [2010,p.46] describe a set of different dimensions to classify environments. As one could guess, one dimension is *deterministic vs. non-deterministic* while a second one is referred to as *perceivable vs. non-perceivable.* The second one describes, if an agent is able to completely perceive its environment or if it has to cope with uncertainty or incorrect information.

Due to the fact that an exhaustive discussion on software agents is beyond the scope of this work, we refer to the literature. Good introductions into agents and agent system as well a discussions on further aspects can be found in [Weiss, 1999; Wooldridge, 2009; Jennings and Wooldridge, 1998; Bordini et al., 2009; Bradshaw, 1997]. In the next sections, we will take a closer look on several aspects of software agents starting with architectures followed by multi-agent Systems, applications and current research issues.

As mentioned in the introduction of Section 2.4, agents use sensors to receive input from the environment and perform actions to influence the environment. In this chapter, we will discuss the question *How do agents transform inputs to outputs?* Generally, there are two different abstract architectures – *reactive agents* and *state-based agents.* Purely reactive agents will not consider past experiences when choosing the next action. They will solely consider the current input to derive their next step. Contrary, state-based agents keep a kind of world model; knowledge about past experiences or changes in the environment after performing a certain action. When choosing its next step, the agent will consider the current input and its knowledge base. Using such an architecture, agents are able to learn from past experiences and can better anticipate the outcome of a possible action.

In the next subsections, we will take a closer look on both architecture types and discuss assets and drawbacks of existing systems.

2.4.1.1 Reactive Architectures

As mentioned in the former paragraph, reactive agents select their next actions solely on the last input from their sensors – thus they *react* on the last experience. They do not keep a history of past inputs or actions to derive the next action from a chain of past activities. See Figure 2.7. A simple reactive agent may have the task to oversee a specific system parameter and, according to values of this parameter, perform some actions. A good example for a simple reactive agent is an agent that controls the speed of a chassis or CPU fan based on measured temperatures. Due to the fact that the fan speed only depends on the actual system temperature, such an agent does not need any history and as such a reactive architecture is well suited.

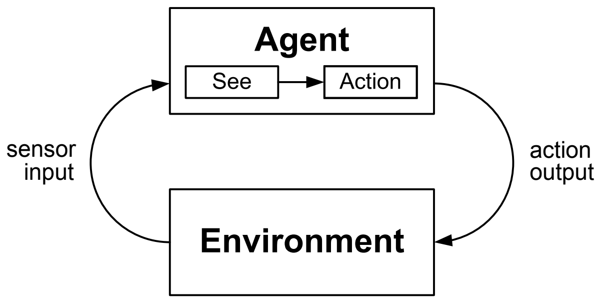


Figure 2.7: Agent and Environment Interaction in a Reactive Agent Architecture. Adapted from [Wooldridge, 1999]

Reactive architectures apply a selection function that usually uses some kind of lookup table to map important inputs to concrete actions. Every other input, that is not related to the agents task, is disregarded.

While reactive agents are rather easy to implement, they have several shortcomings. Reactive agents are considered to have a *short time* view as they only consider the current state and are not able to reason about the environment and possible future evolutions of the system. Furthermore, it is hard to imagine a purely reactive agent that is able to learn form its actions and improves its performance over time. Supporter of this architecture often argue that an overall system behavior somehow *emerges* from the interactions of agents and components in a self-organized way. But emergence is often attributed to complex systems where the relationship between components is hardly, or even not at all, understandable. It seems that such systems, and thus agents making up such a system, are very hard to engineer and one has to use tedious experimentation to create a systems that behaves in an anticipated way.

Kaelbling [1987] gives a good overview of the aspects concerning the development of resource-bounded reactive agents. The best-known reactive architecture is the subsumption architecture developed by Brooks 1986 that is mainly applied for mobile robots. Discussions on alternatives can be found in [Maes, 1990] and [Agre and Rosenschein, 1996].

2.4.1.2 State-based Architectures

In contrast to the purely reactive agents described in the last section, state-based agents determine their next action using past experience and actual input. State-based agents maintain a world model that describes the environment and keeps track of their past actions as well as the perceived outcome of those actions. Furthermore, they have a set of possible actions and some kind of reasoning engine that allows for a mapping of world state and input to a concrete action. By taking into account past experiences during the reasoning process, they are able to learn from their history and can improve their suitability and performance over time. Compare Figure 2.8.

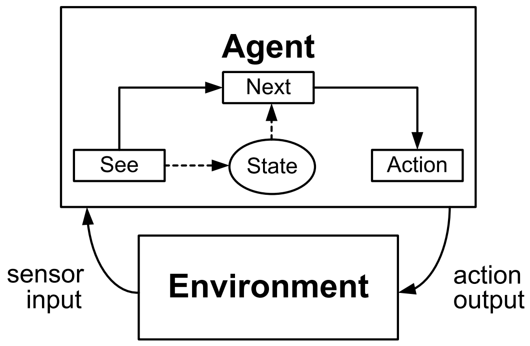


Figure 2.8: Agent and Environment Interaction in a State-based Agent Architecture. Dashed Lines indicate Access to the Agents State and Knowledge Base. Adapted from [Wooldridge, 1999]

The most influential model for state-based agents is the *belief-desire-intention* (BDI) architecture. Based on Bratmans work *Intentions, Plans and Practical Reasoning* [1999], that covers aspects on human reasoning and goal-directed behavior, the BDI model was introduced to the agent community by Rao and Georgeff in 1991. While most reasoning architectures only consider beliefs and desires, e.g. goals, as the important building blocks of the system, Bratman [1999] argued that intentions play a crucial role in goal-directed behavior. They act as partial plans, that are currently pursued and targeted towards achieving one or more of an agent's desires.

The general architecture as introduced by Rao and Georgeff [1991] is structured in the following way. First, every agent has a believe base, that stores the agent's knowledge about its environment and the agent itself. Second, a set of desires or goals is maintained, that contains states of the environment which the agent wants to achieve. Third, as mentioned above, a number of intentions is generated respectively updated during each execution cycle based on the current beliefs and desires. Intentions and the associated plans are commitments to a specific desire. While holding a certain intention, the agent commits to achieve the goals targeted by that intention. During each execution cycle one intention is selected from the set of intentions and afterwards executed. In a deliberation process, e.g. while updating its intention set, the agent can decide to drop intentions and/or a complete desire if it becomes clear, that this desire cannot be achieved anymore or that it has become obsolete.

Continuing his work on BDI agents, Rao developed a logic language named *AgentSpeak* that is tailored to the description – and later to the programming – of BDI-based agents. AgentSpeak was presented in [Rao, 1996]. Considerable research effort has been devoted on AgentSpeak including a version for mobile devices [Rahwan et al., 2003] and an extended version to allow for model-verification of multi-agent systems [Bordini et al., 2003]. One of the latest implementations of AgentSpeak is *Jason* [Bordini and Hübner, 2004] which is implemented in Java and freely available. Beside its BDI underpinning, Jason also allows for a transparent multi-agent distribution across the network, meaning that agents residing on different hosts are able to act in concert. Other systems to mention here are Jadex [Bordini et al., 2009] and JAM [Huber, 1999] whereas Collier et al. [2000] and Busetta and Ramamohanarao [1998] describe systems for mobile networks. We will take a closer look at these two systems in Section 2.5.3, where we discuss architectures for mobile agent systems. Beside research on programming or implementing BDI systems, Kinny et al. [1996] extended well-known object-oriented paradigms to create a methodology and modeling technique for BDI-based systems.

A second architecture for cognitive modeling is *SOAR* [Lehman et al., 1998]. While widely used in areas of social science and psychology, SOAR could not really gain ground in the agent community in computer science, so we refer to the literature for more information.

2.4.2 Distributed AI, Communication and Cooperation

So far, we have mainly talked about agents as a single entity. However, agents are often embedded in a environment with many other agents – so-called *multi-agent systems* (MAS) [Weiss, 1999], which are grounded in the field of *distributed artificial intelligence* (DAI). Despite being a hot topic in computer science, this area also has roots in many other research disciplines as for example artificial intelligence, sociology, economics, organization and management science, and philosophy.

MAS may be an answer to the increasing heterogeneity and complexity of modern computer systems. While each agent is only responsible for a small aspect or task of the whole system, the combination of a large number of interacting, moderately intelligent agents may be able to solve distributed problems thereby showing some kind of emergent behavior. There are several types of interaction between agents. The basic distinction of interaction is cooperation or competition. The first one assumes agents to be altruistic and to work together in pursue of a higher level goal that cannot be achieved by a single agent. The latter one considers agents as entities that pursue their own goals. The aim of the DAI community is to answer the question *when and how will agents interact with each other* in order to achieve personal or shared goals.

Multi-agent systems are often attributed with terms like self-organization or emergence. These attributes derive from the ideas that a large number of agents, where each one is only responsible for a certain aspect of the system, is capable to solve more complex tasks. It is assumed that agents in such a system are able to perform some kind of distributed planning and negotiation on task assignments. But, as mentioned in Section 2.4.1.1, emergence and self-organization are difficult to achieve and to control.

Research topics in the field of distributed artificial intelligence include cognitive modeling, social coalitions, collaboration and competition between single agents and societies of agents as well as interaction protocols and agent communication languages [Kone et al., 2000; Sun, 2006; Rooney et al., 2004].

2.4.3 Methodologies and Agent Oriented Software Engineering

Other scientist are concerned with design methodologies and engineering paradigms. For example, some researchers claim, that object-oriented software engineering (OOSE) is a good advance from older approaches but does not go far enough for agent-based systems. In OOSE, everything is modeled as an object regardless of the kind of entity that is modeled. In an object oriented system, one will thus find *passive* objects, that are mere data closures, for example objects that model a contract or a database table.

Contrary, there will be *active* objects that hold the actual flow of control and determine the actual behavior of the system. This analysis led to the development of Agent-oriented Software Engineering respectively Programming by Shoham in 1993, which is still an area of active research [Huntbach and Ringwood, 1999; Bergenti et al., 2004; Wooldridge et al., 1999]. To support AOSE, researchers have extend well-known OOSE languages and tools like UML to allow for an easier modeling of agents and agent systems [Odell et al., 2000; Bauer et al., 2001]. Critics of this so called Agent-based Unified Modeling Language (AUML) argue, that the language is well suited for the modeling of agent interactions but means for definition of agent properties and agent models are rather limited [de V Peres and Bergmann, 2005]. Beside agent-oriented modelling, [Shoham, 1993] and [Huntbach and Ringwood, 1999] have introduced new programming concepts and languages that better suite for the implementation of agents and agent systems.

A broader approach is taken by researchers, which work on whole engineering methodologies for agent systems. Such methodologies cover the complete specification and design process and provide straight guidelines for engineers and developers. Several methodologies have been developed; each one tailored to specific agent models like BDI. The most notably methodologies for agent system engineering are Gaia [Zambonelli et al., 2003], Tropos [Bresciani et al., 2004], MaSE [Wood and DeLoach, 2001] and Prometheus [Padgham and Winikoff, 2004]. [Bernon et al., 2005a,b] give reviews on existing methodologies and argue for a meta-model approach to create a unified methodology. The biggest drawback of nearly all methodologies is that they do not cover the step from design to implementation. Thus, the mapping from model to code is completely up to the programmer and requires a high amount of knowledge about both – the methodology and the target system. Current research tries to overcome this gap by using a Model Driven Architecture (MDA) approach. A platform-independent model is introduced that allows for the creation of agent implementations independent from methodology and agent system. See for example [Amor et al., 2005]. For further information on methodologies we refer to the literature [Iglesias et al., 1999; Zambonelli and Omicini, 2004; Henderson-Sellers and Giorgini, 2005].

The so called *Agent Modeling Language* [Trencansky and Cervenka, 2005] is an extension to the UML 2.0 meta-model and tries to combine the best of OOSE and AOSE. It incorporates the most significant concepts of software agent engineering like methodologies (e.g. Gaia, Tropos, MaSE), agent models (e.g. BDI), modeling and specification languages (e.g. AUML, UML, OCL, OWL) and agent platforms (e.g. Jade, FIPA-OS, Jack) and is independent from any particular theory, technology or implementation environment. Due to is grounding in UML, AML can be further extended easily and is supported by CASE tools. The authors tried to address deficiencies of current modeling languages like insufficient documentation, proprietary modeling constructs, tailored at modeling of only a limited aspect of a system, applicable only in a specific domain, theory, architecture or technology. But this huge aim leads to the major drawback of AML: it has become very big and complex. It is doubtful if one can model every aspect of a agent system where, for example, emergent behavior is desirable but cannot be anticipated in advance. Furthermore, it is doubtful if such a complex framework, where a single engineer would not be able to understand every detail of a huge model, can really help to ease development.

2.4.4 Systems And Applications

This section covers systems and applications that apply agents in one or more aspects. For several years now, the term *agent* has evolved to a buzzword widely used for all kinds of applications. We are surrounded by mail agents that sort our emails, agents looking like dogs or wizards that help us to write a letter or create a presentation. So called *user agents* provide an interface to a (remote) system accepting requests and instructions and display results. Auction agents take your part in overseeing an online auction and bid until they reach a given limit or win the auction. Nevertheless, most of these agents are rather simple or do not fully comply to our definition of agenthood. Therefore, we will only take a look on systems, that fall into our concepts. Most of these system, e.g. air traffic control or production coordination, apply a multi-agent approach where agents work together and pursue a number of shared goals like reduce number of crashed airplanes or maximize the number of goods produced per day. Kinny et al. [1996] describe an air traffic control system supported by BDI agents. In [Van Dyke Parunak, 1987], a multi-agent manufacturing system named YAMS is described where agents collaborate using the Contract Net protocol [Smith, 1980]. As listed in the introduction of [Weiss, 1999] or in [Fasli, 2007], other examples of multi-agent applications are electronic commerce and markets where buyer and seller agents trade commodities on behalf of their owner. Further examples for agents in eCommerce and supply chain management can be found in [Al-Jaljouli and Abawajy, 2010] and [Zimmermann et al., 2006]. Real-time monitoring and management of telecommunication networks [Manvi and Venkataram, 2004] is another area where agent systems have shown to be valuable. We refer to [Jennings and Wooldridge, 1998] for a more elaborate discussion of multi-agent systems and applications.

Bordini et al. [2009] classified available agent systems into two categories: Logic or process algebra based systems and Java based systems. This distinction considers the languages that are used to describe agents and not the language of the underlying system that executes these agents – which is Java for all presented systems. Namely, the systems that fall into the first category are Jason [Bordini and Hübner, 2004], 3APL [Hindriks et al., 1999], IMPACT [Subrahmanian et al., 2000] and CLAIM/SyMPA. All four systems have in common that they adapt BDI-like concepts, e.g. plans, beliefs and goals, as well as a deliberation cycle to reason about future actions. Only Jason implements an extended version of AgentSpeak [Rao, 1996], and thus the original BDI model.

JADE [Bellifemine et al., 2007], Jadex [Bordini et al., 2009] and JACK [Bordini et al., 2009] are the systems presented in the second category. Here the underlying systems as well as agents are written in Java. JADE is one of the most widely used agent platforms initially developed by TILAB[9], which still holds the copyright. By now, it is open source and in 2003, the JADE board was founded to supervise the development of the project. Currently, the JADE board has three members, namely TILAB, Motorola[10] and Whitestein Technologies AG[11]. JADE is FIPA-compliant[12] and offers a rather simple agent model that serves as the basis for the development of more sophisticated agents. A retrospective on the Jade system and its achievements can be found in [Bellifemine et al., 2008].

Jadex [Pokahr et al., 2005] is an extension that introduces the BDI model [Rao, 1996; Walczak et al., 2007] for agent development into the Jade system. In contrary to the systems in the first category, Jadex BDI agents are made up from an XML-based belief, goal and plan descriptions accompanied by Java classes that implement plan behavior. As Jade, Jadex is available as open source. Over the past years, Jadex has evolved into the *Jadex Active Components* [Pokahr and Braubach, 2009; Pokahr et al., 2010] middleware which provides a managed execution environment for active components. In contrast to traditional (passive) components, active components exhibit a certain amount of autonomy regarding their execution. Instead of just reacting to requests, they can actively decide to perform some actions. With the evolution of the whole platform into an active component middleware, the agent part became an extension

[9]http://jade.tilab.com
[10]http://www.motorola.com/
[11]http://www.whitestein.com/
[12]http://www.fipa.org

to the new system. A second extension is *Jadex Processes*[13] which provides execution facilities for BPMN- and GPMN-based workflows [Leymann and Roller, 2000].

Another system, that integrates workflows in an agent system is WADE [Caire et al., 2008b]. Similar to Jadex, WADE extends the JADE agent system with the ability to execute workflows. Workflows can be created using an Eclipse-based graphical editor called WOLF [Caire et al., 2008a] and are afterwards exported to Java code. There are several kinds of actions that can be added to a workflow, e.g. to control the flow of execution or access remote services. Even a container action that can be filled with custom Java code is available. A single class file is created for every workflow and is executed by special workflow agents. Due to the fact that the mapping between workflow and code is rather complex and imposes various constraints, using the system requires profound knowledge. Thus, it is clearly targeted at experts and not at end users. Interesting to note, the authors claim that, with the mapping to Java classes, they have introduced inheritance to the workflow metaphor as one could use an existing workflow class and extent it to create a more specialized version.

In contrast to JADE, JACK [Padgham and Winikoff, 2004] is a commercial agent system developed by the Agent Oriented Software Group. JACK is based on the BDI model too and offers an agent-oriented language to describe agents using common BDI constructs like goals, plans and beliefs. Other components provide standard agent system functionality like message exchange support or a name server to find other agents. Beside that, JACK comes with a number of tools like a graphical plan editor to support the development of agent-based applications.

A third section in [Bordini et al., 2009] covers industry specific applications, namely DEFACTO [Bondalapati et al., 1999; Schurr et al., 2005] and ARTIMIS [Sadek et al., 1997], and we refer to the appropriate literature for further information.

Actual research on software agents or multi-agent systems is concerned with modeling human cognition and social interaction and their mapping into multi-agent systems [Sun, 2006]. This research area is mainly grounded in cognitive and social science, but to some extend depends on multi-agent systems to help in modeling and testing theories, perform experiments or reassure real-live experiments. Another area of high activity are interaction protocols for agents [Agre and Rosenschein, 1996; Labrou et al., 1999], that describe the way, how agents should behave in a conversation with each other and how they should share knowledge. There are several extends that are of interest depending on the kind of relationship between conversation partners, e.g are they working together

[13]http://jadex-processes.informatik.uni-hamburg.de

or is each one pursuing its own goals. Due to their heterogeneous and open nature, it is hard to predict, how agent systems will behave, how fast agents are able to solve specific tasks and if they will be stable in the long run. Thus, research and advancements on reliability, stability and traceability of agent systems are important for a wider acceptance of agent-based systems.

Over the past years, Grid and Cloud computing have gained reasonable attention from research and industry. The agent community started to exploit the possibilities when incorporating agent technology into Grid and Cloud systems. One hot topic is the transparent reconfiguration of a cloud system regarding the location of data and computation. In [Chaimontree et al., 2011], an approach to manage a large cluster of machines is presented where agents represent single cluster entities to coordinate the clusters runtime behavior. Dynamic resource reconfiguration of a Cloud system is described in [Kim et al., 2011b,a] that allows for a better adaptability and Quality of Service (QoS) of the whole system with respect to current runtime parameters like usage load, number of different users and applications as well as data access schemes. [Hegazy et al., 2010] are especially concerned with an agent-enhanced organization of storage in a Cloud. Other topics which are tackled by agents are monitoring of cloud services as well as ensuring the systems QoS parameters [Peregud et al., 2011; Liu et al., 2010; Cao et al., 2001], service composition [Gutierrez-Garcia and Sim, 2010] and service management [Zhou et al., 2010b; Lopez-Rodriguez and Hernandes-Tejera, 2011; Kim, 2006], information retrieval [Chang et al., 2011] and general system management [Tveit, 2002]. Further aspects on agents and Grids respectively Cloud systems can be found in [Foster et al., 2004].

Supporting team workers in the field by letting agents coordinate their interactions and provide a proper surrogate for negotiations if the actual team member is currently unavailable is an interesting approach to ease the work of mobile workers. [Lee et al., 2007] deployed such a system in the telecommunication industry whereas [Mercadal et al., 2011] have targeted crisis management in emergency situations. An approach to improve organ transplant management using interacting agents is described in [Calisti et al., 2003].

A comprehensive overview on languages and platforms for agent development can be found in [Bordini et al., 2006], whereas [O'Shea et al., 2011] gives a good overview on current research topics. A review on industrial agent systems is presented in [Van Dyke Parunak, 2000], and, with a special focus on traffic and transportation systems, in [Chen and Cheng, 2010]. A simulation of production planning using agents was executed in [Hodík et al., 2005].

2.5 Mobile Agents

Mobile Agents have been introduced as a design paradigm for distributed applications. They are, simply speaking, a combination of software agents and mobile code [Fuggetta et al., 1998; Ghezzi and Vigna, 1997; Carzaniga et al., 1997]. Beside the characteristics of a software agent, mobile agents are able to move freely across the network from one execution platform to another. This process is called *migration* and it is initiated by the agent itself. The agent decides where and when to move. In contrast to mobile code, which is only transferred from one platform to another, mobile agents usually migrate several times thus performing some kind of round-trip or star-shaped itinerary. This ability is often referred to as *multi-hop*.

During a migration, the execution of an agent is stopped and its current state is preserved. Afterwards, the agents code and state are transferred to the destination platform where the state is restored and execution will continue.

What at first sounds like a kind of computer virus – an entity moving freely across the network – is in most cases much more good-natured. There are several aspects in the design of mobile agent toolkits that are targeted to prevent an ill-natured usage of agents. First, every agent is executed inside a so-called *agency* which can be seen as a sandbox. The agency controls an agents life cycle and limits its abilities while at the same time providing services and access to legacy systems. Thus, mobile agents cannot move freely between any different host. They are merely bound to migrate between different agencies, whereas it is possible that a single, physical host runs more than one agency.

Second, most mobile agent systems implement several security mechanisms to provide fine-grained access to different aspects of the system. Some systems, like Semoa [Roth, 2001], are nearly exclusively focused on security and grouped all remaining system parts around a complex security core. For example, mobile agent toolkits written in Java – and that are currently the most – use a class loader hierarchy to control class and object visibility thereby removing possible points for harmful actions. But with the overwhelming power of an agency comes another security problem – that of a malicious agency. Due to the fact that an agency controls the complete life cycle of an agent and is in charge to load its code and instantiate its classes, an ill-natured agency is able to alter an agent's code or data, provide false information to an agent or contradict its execution in some other way. As can be guessed, protecting an agent from a harmful agency is a much more challenging task than to secure an agency against malicious agents. We take a deeper look into mobile agent security in Section 2.5.2.

According to [Lange, 1998; Chess et al., 1997] mobile agents offer several benefits:

They reduce network load. Distributed applications often suffer from unnecessary high network load caused by communication protocols that are not well-suited for the desired functionality of the remote client. A remote invocation may lead to multiple requests and responses that get more and more specific during this interaction. A second problem are huge replies due to unspecific requests where a large portion of a reply is just thrown away after minimal examination. In summary, unnecessary network traffic is caused just to compensate the limitations, specifically the generality, of the protocol. Instead, a mobile agent can encapsulate such an interaction and perform it locally at the remote system. And it is able to filter replies at the remote host and will afterwards only transfer the useful data. Thus, it will not produce any network load during the interaction with the legacy system and will further reduce the network load due to the reduced size of the results that are sent back to the home system. This concept is often described with *code shipping instead of data shipping.*

They overcome network latency. Many systems, as for example industry production processes, need real-time responses to changes in their environment. Especially in larger networked architectures, this becomes a challenging issues due to network latency which imposes a random delay to each remote communication. Using mobile agents, which are able to operate on a remote host, these delays can be omitted.

They encapsulate protocols. This point is somewhat related to the first. Due to their ability to act as client representatives on the server side and perform all the interaction, mobile agents encapsulate task or application specific protocols. A distributed application, that is based on a mobile agent architecture, just needs a single protocol for remote communication: the migration protocol for its agents. Any other protocol can be encapsulated inside an agent. Beside the fact that the network layer of such a system is relatively easy to implement, it is also very easy to introduce new functionality by simply deploying new agents.

They execute asynchronously and autonomously. Due to their ability to change the execution platform and act autonomously from the system where they have been started, mobile agents are a natural choice for applications that operate in unreliable and throughput-constrained networks. As for example, mobile users often use such network connections, which are, additionally, sometimes very expensive.

Here, traditional applications that demand a permanent network connection are all but optimal and mobile agents seem to be a good alternative.

They adapt dynamically. Mobile agents, as a special kind of software agents, are able to sense their environment and react in a timely fashion to changes. For example, a group of mobile agents can distribute itself among a number of platforms to achieve an optimal configuration to solve a given problem. Or agents may decide to change the current host due to performance issues like CPU or memory overload.

They are naturally heterogeneous. Networked systems are naturally heterogeneous, both in hardware and software. Mobile agents, which are independent from those differences due to their own, uniform execution environment, provide a good solution to cope with this heterogeneity.

They are robust and fault-tolerant. Using their ability to move from host to host, mobile agents can adapt to unforeseen evolutions like a host going down. Before shutting down, the host could inform all agents running on this machine to leaving the system and continue their execution on another host.

Another advantage, stated by Johansen et al. in 1995, is that with using mobile agents, you do not need to maintain a distributed state as with a client server architecture. Agent and state form a unit and the state is always at the same place where the agent is. Thus one does not need to cope with several parts of a single state that are distributed across the network, e.g. one part at a server and several parts on a number of clients or vice versa.

Based on these advantages, a number of application scenarios seem to be well suited for the adoption of the mobile agent paradigm. These include electronic commerce, personal assistance, distributed information retrieval, information dissemination and workflow applications and groupware, just to name a few. Nevertheless, critics often ask for *the* killer application and so far, none could be found. All the above mentioned applications can be realized using more traditional and well-known techniques like client server. But compared to other industries, as for example automotive, many techniques never delivered a killer application but are widely used today. We could still have our cars pulled by one or two horses, but in times of petrol engines or hybrid electric power trains, who would actually care about administering a horse. Those new engines did not allow for a whole new type of movement. They just introduced a new way to handle the movement.

While other approaches may be better suited for a certain aspect of an application, Chess et al. [1997] stated, that no other concept unifies the above mentioned benefits better than mobile agents. Even if mobile agents do not introduce new application types, they have theoretically proven to allow for more robust, scalable and simpler solutions to a number of applications.

Unfortunately, research interest in mobile agents has dwindled over the last few years. The reasons are manifold; the lack of a standard concerning agent or platform models and architectures as well as open research issues like security made many researchers skip that topic. For example, [Roth, 2004] described some obstacles, that prevent a wider adoption of the mobile agent paradigm. Nevertheless, we believe that it is still a very interesting and promising concept and the research community should rather address open problems instead of skipping fascinating ideas. A good overview on the general principals of mobile agents and applications based on this concept can be found in [Genco, 2008].

In the next sections, we will take a closer look on several aspects of the mobile agent paradigm. We will first look at the migration process in detail followed by a description of architectures and systems. Afterwards, we outline some application scenarios and take a survey on open research issues and the current research situation.

2.5.1 Migration Process

The ability to change their execution environment during runtime distinguishes mobile agents from other kinds of software agents. Thus, the migration process is the most interesting part for research and development and since the dawn of mobile agents, a lot of work was conducted to increase migration performance. First, we will outline the steps that make up a single migration and compare *strong* and *weak* agent migration. Second, we will outline the evolution and current state of the art in agent migration. In the following section, we will have a look at architectures, applications and current research topics.

What sounds rather easy – move from host A to host B and continue execution – involves several, partly complicated steps. Taking a closer look at the whole process, we can identify the following steps (compare figure 2.9):

Capture execution state As soon as the agent decides to migrate to another host, the underlying agent platform should stop the agent's execution and capture the agent's execution state, e.g. runtime stack, values of local variables, and so on.

Pack execution state and code After the state is preserved, the state and code must be made ready to be send to the other host. This usually involves some compressing and flattening code and state into a byte sequence.

Transfer over the network This byte sequence is afterwards transferred across the network to the remote host. After a successful transmission, the above mentioned states are executed in reverse order.

Unpack execution state and code On the remote host, the byte sequence is processed and the agent's state and code are restored.

Restart agent After having unpacked the agent and its code, the execution of the agent is resumed.

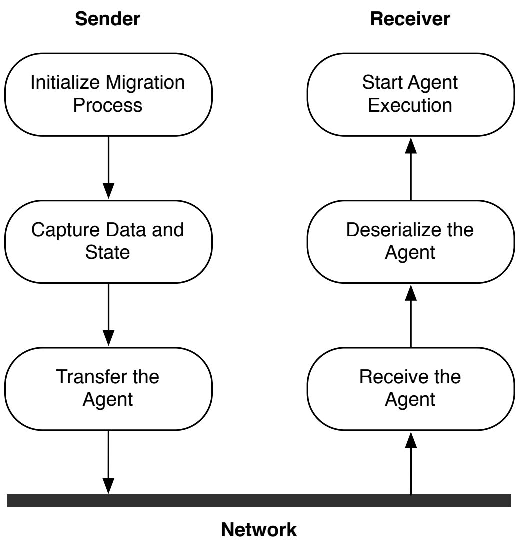


Figure 2.9: Agent Migration Process.

The two most critical and complicated steps of the five presented above are capturing and restoring the agent's execution state. According to the implementation of these two steps, weak and strong migration are distinguished. We will start with the discussion of strong migration as the first mobile agent toolkits all supported this migration type. With strong migration, the complete migration process is encapsulated in a single command – usually a *go* statement. Using that command, the agent announces its desire to migrate and specifies the destination. In doing so, the agent initiates the whole migration process and is transferred to the desired host. There, execution is continued right after this *go* statement. Having an agent system that supports strong migration, the development of mobile agents becomes rather easy as the migration process is totally transparent to the programmer. However, this advantage also comes at a cost. Due to the complete encapsulation of the migration process, the agent or programmer has no way to influence the migration and how it is handled. One must rely on the implementation provided by the underlying system. As mentioned above, most of the first mobile agent toolkits supported strong migration as for example Telescript [White, 1999], AgentTCL [Gray et al., 1996a] or ARA [Peine and Stolpmann, 1997].

One could say that weak migration is the opposite of strong migration as the advantages and drawbacks are nearly exchanged. Using weak agent migration, the system can only capture parts or nothing at all of an agent's state. Instead, the agent respectively agent programmer has to keep track of the agent's internal state before initiating a migration. After the migration was performed and the agent restarted, the execution usually continues at a predefined point, where the previously saved state can be restored. Constructing agents that rely on weak migration usually leads to state machines that control the agent's live cycle. Before a migration, the agent sets a new state and according to this state, the execution is continued at the remote host. As mentioned at the beginning of this paragraph, the advantage and disadvantages of strong and weak migration are reversed. Due to the fact that agent migration is not enclosed in a single statement, programming of mobile agents becomes a bit more difficult. Nevertheless, due to an increased control over the migration, the programmer can much more influence the migration process what introduces new ways for optimizations. There are at least two different reasons for the adoption of weak migration. First, one decides actively for using weak migration to allow for better influence on the migration process. Second, the underlying programming language makes it hard, if not impossible, to use strong migration. As Acharya et al. [1997] showed with their Sumatra system, capturing the state of a thread in Java is impossible without changes to the Java Virtual Machine. Due to the fact that Java has become the language of choice for implementing mobile agent toolkits, many of them, like Aglets, Mole [Baumann et al., 1998], Jade [Bellifemine et al., 2007] or Voyager [Kotz and Mattern, 2000], only support weak migration.

In the last paragraph, we mentioned that, when using weak migration, the agent respectively the programmer is able to customize the migration process. We now give a short overview on actual migration strategies and outline the points, where one is able to configure a migration. With the rise of Java and the amount of mobile agent system continuously rising, two main migration strategies evolved in the late 90s respectively the beginning of the current millennium. These two are referred to as *Push-All-To-Next-Agency* and *Pull-From-Home-Agency*; the names roughly describe the main approaches for migrating an agent's state, code and data. The first strategy, Push, simply packs agent state, code and data into a single container and sends it at once to the desired host. There, everything is unpacked and execution is resumed. As one can see, this strategy fits nicely with the autonomy property of software agents. As soon as the agent is transferred to the remote host, no link back to the last agency is needed as all code and date required by the agent has been transferred. A second advantage is the rather easy implementation of this strategy. But all this comes with a price. There may be times when not the complete code or data is needed at the destination agency and those unused parts are transferred superfluously and network load is increased unnecessarily. Using the second type, a Pull strategy, one can omit to transfer unused parts of the agent's code or data as this strategy only transfers the agent's state to the destination agency in the first step. There, the new agency tries to resume the execution of this agent and will load all parts of the agent's code and data on demand. What sounds like an improvement over the simple Push strategy introduces a new point of failure. The destination agency will need a constant uplink to the last host or at least a fast way to reconnect. Only under those conditions the agency will be able to load code and data as needed. If such a link is not present or cannot be established, the agent's execution will terminate as important parts are missing. This limits the autonomy of a mobile agent in critical way. Nevertheless, a Pull strategy is the better choice if one tries to keep the network load as small as possible.

Based on these strategies, Braun and Rossak [2004] developed a migration engine that provides much more options to configure a migration. Braun and Rossak could show that neither of these two strategies performs best in all cases and they argue for adaptable migration strategies that can be configured by an agent during runtime. This thesis led to the development of *Kalong*, a virtual machine for agent migration. With Kalong, the agent respectively the programmer is able to configure a migration in various aspects. For example, one can specify a set of classes that should be migrated to the desired host in the first place while all the remaining classes will be loaded on demand. The same can be done for data that accompanies an agent. Data can be packed into well specified blocks and for each block one can defined when and how it should migrate. Thus, Kalong provides a representation of nearly any migration strategy that fits between the extreme Push and Pull strategies. Beside that, Kalong provides some other ways to increase migration performance. First, Kalong comes with a class cache that is able to actively prevent class loading over the network if that is unnecessary. Second, an agent is able to establish several code and a single mirror server. Code servers allow for the agent to store code for later usage. For example, the agent migrates into a subnet that has a very bad network connection to the outside network. In such a case, it would be wise to install a code server inside this subnet. Afterwards, the agent is able to load code from this server on demand and is not bound to load code from its home agency using the slow and error-prone network connection. Kern and Braun [2006] showed that with using code servers in subnets, the migration performance can be increased significantly. We should mention here that an agent can initiate as much code server as there are agencies inside the network[14]. In contrast to code servers, an agent can only install a single mirror server. If the agent intends to install a new mirror, then a probably existing old mirror server has to be released first. On a mirror server, an agent cannot only store code but data too. Thus, the agent is able to deposit currently unused data and decrease the size of data that migrates to the next agencies. Later, if the agent will need that data again, it can load it from the mirror server. It is even possible to up- and download data remotely to and from the mirror server.

Even with the capabilities that Kalong provides, agent migration is still not optimal in cases where the code granularity is not fine enough. When using standard Java techniques for object serialization, the smallest code part, that can be transferred alone, is a Java class. During our work on mobile agents, we found that the code of an agent is often very coarse-grained and consist of only a single or a few classes. Thus, most of Kalong's features are useless – if only a single class is transferred, there is not much room for fine-tuned migrations. Based on these experiences, we started to work on a class splitting technique, similar to [Krintz et al., 1999], that allows for the separation of existing Java classes on the Bytecode level. We were able to divide a single classes into several new classes by moving methods from the original class into newly created ones. The distribution of methods among the new classes was grounded on execution probabilities derived from static code analysis and profiling runs. In [Kern et al., 2004], we presented experimental results that show a significant increase of migration performance when using the new fine-grained code set instead of the original one. Further ideas to model the migration process of mobile agents can be found in [Xu and Qi, 2006].

[14]We do not claim that this is a successful or even fast strategy. We only state, that the agent could do so.

2.5.2 Security

When compared to traditional paradigms, systems that rely on mobile code, that moves across an open, potentially untrusted networks, introduce a set of new security issues [Hohl, 2001]. [Chess, 1998] lists the main concerns that must be tackled in order to secure a mobile code/mobile agent system. First, mobile code that entered a system may come from an untrusted host. Thus, the current host, that is going to execute the mobile code, should be protected against any harm that may be caused by the foreign program. For example, the program may try to masquerade as a known and trusted user in order to execute critical system commands or to access the file system. It may even clone itself and distribute its siblings to other known machines (commonly referred to as *worm*). The easiest solution would be to restrict the network to a set of trusted hosts which would allow for executing code that comes from another trusted host. And, in case of a harmful action, one could track down the originator of the malicious program. But this scenario contradicts the general assumption of a world-wide heterogeneous network where hosts and mobile code respectively agents enter and leave at random. Thus, the originator of an agent is often unknown, so we need techniques like digital signatures to build up a trust level.

A second issue, that is much harder to solve, is the protection of mobile agents from malicious hosts. Consider an agent that roams the network in search for the cheapest price of a product. On each host it visits, the agent gathers the actual price and moves on to the next vendor. A malicious host could try to alter these so-far collected prices or adapt its own offer in order to be the cheapest vendor. Such an attack is sometimes describes as *brainwashing*. Another form of attack is called *hijacking*, where an agents code is altered to influence the behavior of that agent. A vendor's host could change the price-search agent in so far that this agent will only consider the vendor's offer. Due to the fact that the host is in complete control of the executed program, many researchers argue that this problem cannot be solved. [Sander and Tschudin, 1998] describe a number of threats that can arise and suggests several techniques to handle these threats. Shen and Tong could show in 2009 that the security level of a mobile agent system can be improved significantly by introducing a trusted third party that handles and controls interactions and data as well as agent exchange.

As stated above, we cannot rely on the underlying operation system (OS) to enforce security policies due to the fact that mobile agents roam in heterogeneous networks with different OS. Features of one OS may not be provided by another one and we can only take a small set of features that any OS supports for granted. The most promising approach to tackle this issue is the usage of virtual machine language to write mobile code and an interpreter for mobile code execution [Volpano and Smith, 1998]. The interpreter sits on top of the OS and can enforce security. This is another important reason why Java has become the language of choice for implementing mobile agent systems. Beside being an interpreted language, Java comes packed with a great set of security mechanisms [Oaks, 2001].

2.5.3 Systems and Applications

The first mobile agent system was *Telescript*, introduced in 1994 by White. Telescript is an interpreted, object-oriented language with a collection of hierarchically organized classes. It offered build-in autonomous process migration, which means that agent execution after a migration continues with the next statement right after the migration statement. From the programmers point of view, this is the most elegant way of writing mobile agents as the system controls the whole migration process. With encapsulation of the migration process into a single statement, the programmer is free to write agents that move around the network without bothering about the details. Later systems, e.g. those written in Java, demand a higher effort from agent developers as the migration process is at least partly controlled by the agent itself. Telescript did not introduce any build-in security mechanisms to protect agents or platforms. Thus, security is a programmer issue that often leads to *paranoia programming*. Trying to secure every line of code is tedious during writing a program and results in systems that are unnecessary complicated, hard to maintain and error-prone. Further discussions on Telescript can be found in [Tardo and Valente, 1996] and [White, 1999].

Another project that should be mentioned as one of the first focusing on mobile agents is *TACØMA* [Johansen et al., 1995]. The main goal was to provide software support for mobile agents and for about seven years, a number of prototypes have been developed to tackle different research issues. None of these prototypes actually evolved into a real application; the team rather focused on learning from experiences with one prototype and then moved on to the next problems.

TACØMA introduced an interesting concept for an agent model and agent migration. Each agent has its own briefcase, that contains the data and code associated with that agent. Several so-called file cabinets exist on every host, which consist of a number of folders. These folders can be used by agents to store and retrieve data, which is not used at the moment. Even complete agents can be stored in a file cabinet as the agent's briefcase is just a special kind of folder. Early versions of TACØMA offered a taxi service for agent migration; a single agent carrying several briefcases moved from host to host to deliver those agents to their desired destination. At the destination, a briefcase is handed to gateway agents, which decide on next steps concerning the new arrival. In later versions, each briefcase was directly transferred to the destination.

All TACØMA systems are consider as a kind of glue for composing programs and not as a full-fledged environment to implement whole applications. The TACØMA workgroup chose to implement weak migration in their systems to allow for complete control over the migration process, i.e. what makes up the current state of the agent and what parts need to be transferred. Beside agent migration, each new prototype provided one or more new features like multiple language support, synchronization techniques between agents, wrapper agents to encapsulate legacy systems and potentially malicious agents or had a smaller footprint for execution on mobile devices. As an example, [Sudmann and Johansen, 2000] could show that, by using agent wrappers, a WebCrawler, that first migrates to the remote host can operate significantly faster than a Crawler working remote. For a complete overview on the whole project we refer to [Sudmann, 1996] and [Johansen et al., 2002].

The first mobile agent systems where mostly based on scripting languages like Tcl [Ousterhout, 1994]. Beside TACØMA, systems like Agent Tcl [Gray et al., 1996a,b], D'Agents [Gray et al., 2002] or ARA [Peine and Stolpmann, 1997] were based on Tcl or allowed for the execution of agents written in Tcl. A scripting language offers several advantages; the most influential one is, without doubt, that agent migration and platform independence can be easier achieved as with a compiled language like C. Beside that, scripting languages are usually easy to learn and lead to faster results due to a much shorter feedback and development cycle. Offering a library for GUI programming named Tk, Tcl also allowed for rapid user interface development. Later, as Java became more and more the language of choice for networked applications, Java-based mobile agent toolkits like Mole [Baumann et al., 1998], Aglets [Lange et al., 1997], Grasshopper [Magedanz et al., 1999], Jade [Bellifemine et al., 2001] or Tracy [Braun et al., 2001, 2005] evolved. Java offered a number of advantages over the so far used scripting languages. First of all, it is truly platform independent due to the application of a virtual machine that is responsible for the execution of Java byte code. Thus, agents as well as agent platforms written in Java can be executed on every host that provides a *Java Virtual Machine* (JVM). Additionally, agent migration can be implemented using language features like object serialization for state capturing and class loaders for loading code from remote hosts. However, as mentioned in Section 2.5.1, using the standard JVM one is bound to weak migration as capturing the execution state of a Java thread is impossible without changes to the JVM.

As described in 2.4.1.2, the BDI architecture is the most widely used architecture for state-based agents. Therefore, it is not a surprisingly that several research groups worked at developing mobile agents which are based on BDI. Aim of all those project was merely to have the power of beliefs, desires and intentions in the mobile agent world and thus, most of the groups just made BDI agents mobile while not taking into account the specifics that only occur in the world of moving code. Many disregarded the fact, that an agent's code should be as small as possible to keep the migration overhead low. Mobile BDI agents presented so far are merely ordinary BDI agents transferred over the network.

Other systems to name here that support agent mobility are JAM [Huber, 1999], JADE/Jadex [Bordini et al., 2009] or a mobility supporting AgentSpeak implementation [Rahwan et al., 2003]. Beside Jadex, mobility and BDI are also combine in [Collier et al., 2000] and [Busetta and Ramamohanarao, 1998]. Other approaches, for example process model based agents presented in [Paulino et al., 2003; Paulino and Lopes, 2006] or XML-based agents [Steele et al., 2005], have been evaluated. For a good state-of-the-art summary on mobile agents, we refer to [Gray et al., 2000].

Application domains that seem to ask for the usage of mobile agents are Information Retrieval [Covaci et al., 1998; Brewington et al., 1999], Data Mining [Moemeng et al., 2009; Yubao and Renyuan, 2009; Kulkarni et al., 2007; Chang et al., 2008] and Distributed Resource Information Management [Dale, 1998]. Information Retrieval and Data Mining involves agents that roam the network to search information, filter it and deliver only the relevant results to their owners. The counterpart of Information Retrieval is Information Dissemination, where agents spread information across the network by moving from host to host. Both applications can be seen as the foundation of all other scenarios, where agents collect or spread information among the network. For example, [Brugali et al., 1998; Kern et al., 2006a] describe, how mobile agents can be utilized in supply chain management to coordinate interactions across several partners that are involved in the same business process. [Di Caro and Dorigo, 1998] describe a routing algorithm for telecommunication networks that is based on mobile agents. [Lange, 1998] lists several application scenarios that seem to be, or have proven to be, well-suited for the mobile agent paradigm, for example electronic commerce [He and Leung, 2002; Fasli, 2007; Pathak et al., 2009; Autran and Li, 2009; Kowatsch et al., 2008; Hou, 2009], personal assistance [Kern et al., 2006b], workflow applications [Feng and Cai, 2008], e-Learning [Wang et al., 2009], resource sharing [Suna et al., 2004], telecommunications

[Van Thanh, 2001], image retrieval [Picard and Cord, 2006] as well as climate analysis and prediction [Ioan and Liliana, 2008].

Like multi agent systems in general, mobile agents have been applied to the management and maintenance of Cloud and Grid structures as Foster et al. suggested in 2004. The available results range from full-fledged cloud and grid systems based solely on mobile agent technology [Chen et al., 2010; Zhang and Zhang, 2009; Aversa et al., 2009, 2006] to specific tasks like job coordination [Fukuda et al., 2006] and intrusion detection [Dastjerdi et al., 2009]. Beside Cloud and Grid systems, others have focused on providing a mobile agent based middleware for distributed and wireless applications [González-Valenzuela et al., 2011; Aversa et al., 2003; Chang and Fan, 2010; Raza and Shibli, 2007].

One of the largest application area for mobile agents is network management and maintenance as well as establishing ad hoc and sensor networks. The ability to inject new agents that are capable to freely roam the network, find their path across all nodes and alter, for example, their configuration proved to be very valuable. Thus, a lot of research effort has been devoted into different aspects of this topic. [Yamaya et al., 2004] describe a system that is capable to establish ad hoc peer-to-peer network using mobile agents whereas [Herrmann, 2003] focus on creating general logical networks above the physical layer. Different means to spread data inside a network are covered in [Chen et al., 2007b] and [Lu et al., 2009] and in [Massaguer et al., 2006] techniques to explore wireless sensor networks using mobile agents are discussed. Such methods are always governed by energy considerations [Arai and Sugiyanta, 2011] as well as optimized routing algorithms [Chen and Zhang, 2009; Manvi and Venkataram, 2007; Wu et al., 2010]. A huge amount of work has been devoted to the programming of networks [Aiello et al., 2011; Chen et al., 2006; Szumel et al., 2005; Tong et al., 2003] and to the creation of middleware concepts that easy the development and deployment of networks [González-Valenzuela et al., 2010b,a; Shen et al., 2009]. Another important issue in network maintenance is security and mobile agents are applied especially to network intrusion detection [Wang et al., 2006; Xu and Li, 2009; Patil et al., 2008] or trust management [Yeager and Chen, 2007]. A general overview on mobile agents in network management is given in [Bieszczad et al., 1998; Ranganathan et al., 1997; Chess et al., 1995]. Furthermore, special considerations for wireless sensor networks can be found in [Dagdeviren et al., 2011] and [Chen et al., 2007a]. A survey on latests application areas for mobile agents can be found in [Outtagarts, 2009].

As mentioned in the beginning of chapter 2.5, research interest in mobile agents has dwindled over the last years due to several reasons. First of all, the absence of a killer application made researcher doubt the whole concept. Second, the lack of standardization of agent models, toolkits and methodologies resulted in a wide variety of systems that are unable to interact with each other. Moreover, research results cannot transferred easily between different working groups. The FIPA standardization efforts did not have the desired success. Only a handful of systems, like JADE, are FIPA-compliant. The main reason is surely the complexity of the whole specification, which makes the implementation of smaller systems or prototypes rather expensive. Last but not least, most researchers doubt that mobile agents will ever become a secure technique. As stated in section 2.5.2, no system will every be 100% secure and the same applies for mobile agents. The risks of that paradigm may be higher than of other systems, but can be handled as described in the last section.

Nevertheless, several groups continue their work on mobile agents or introduce the concept into other areas. Still, work is conducted on improving the migration and overall tour performance of mobile agents, for example by applying genetic algorithms on tour planning [Cai et al., 2010; Schlegel et al., 2006], generating online network maps [Erfurth, 2004] or creating content-specific itineraries [Ota et al., 2010]. Other topics include tracing and controlling of mobile agents [Baumann, 2000], multi-task scheduling [Liu et al., 2008] as well as the establishment of a reliable inter-agent communication layer [Choi et al., 2010; Cao et al., 2004; Deugo, 2001].

Some groups argue for a special programming and modeling language as well as a development methodology for mobile agent development that should provide constructs and tools needed to ease the effort of system development [Ledoux and Bouraqadi-Saadani, 2000; Kendall et al., 1998; Kendall, 1999]. For example, UML does not provide useful means to model code and data distribution among a network of hosts. However, others object that it is to early for proposing such comprehensive things as we should first target more low-level problems. Nevertheless, moving towards standardization is one of the issues the agent community has to tackle in order to interest a wider audience for this fascinating paradigm.

2.6 Tracy 2

This chapter will introduce the reader to the Tracy 2 Mobile Agent System actively developed at the Friedrich Schiller University Jena (FSU), Germany [Braun et al., 2005] and it is the successor of Tracy, the first mobile agent system created at the FSU in 1999. Tracy featured a monolithic architecture and helped to experiment with and evaluate agent technology, especially mobile agent migration, in the early year of agent research.

The acquired experiences led to the development of Tracy 2 which is based on highly modular architecture and and a very lightweight agent model. In the remainder of this work, we will refereed to the Tracy 2 Mobile Agent System as Tracy. The following sections describe the overall system architecture of Tracy as well as the enclosed agent model.

2.6.1 Architecture And Components

The monolithic architecture of the first Tracy system presented some drawbacks that made the complete redesign of the system a necessity. First, extending the system with a feature required a single, specialized agent that acted as a proxy between Tracy and the new module. This, combined with the second drawback, a very simple but difficult to use agent model made the adaptation and utilization of the first Tracy system in large-scale applications in heterogenous environments a tedious and error-prone task.

The redesign of the system clearly targeted this issue by introducing a lightweight, modular architecture based on a micro kernel that is accompanied by feature-specific, self-contained plugins and agents.

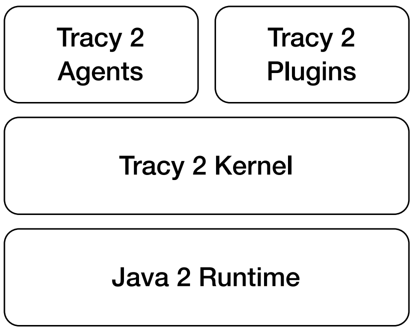


Figure 2.10: Tracy 2 System Architecture

The micro kernel is responsible for the basic system operations such as agent execution and scheduling, hosting of an agency as well as loading and unloading of plugins. Any other feature, like security, message exchange, agent migration or database or web service access is implemented in plugins. Compared to the first Tracy system, plugins can be considered as a feature module with the proxy agent included. Tracy distinguishes between plugin and service – a plugin provides a single service but a single service may be provided by different plugins. This n:m relationship leads to a more fail-safe and robust environment by providing several independent and probably different modules to handle the same task. Every plugin provides a specific context object that acts as the interface to access a plugins functionality and access is controlled by permissions issued to agents and plugins. Context objects are maintained by the current agency and a context object can be requested by either supplying a service or plugin name. To increase the systems flexibility, plugins can be loaded and unloaded during runtime allowing for an easy system reconfiguration while maintaining a high availability.

Tracy comes with a large set of plugins for various needs. Some of the more important ones are the following ones:

Migration To enable agents to switch their execution environment during runtime, the Migration plugin provides weak agent migration in the Tracy system. Due to the fact that agent migration has been the main research focus at the FSU, the Migration plugin provides a very sophisticated migration engine developed by [Braun and Rossak, 2004] which has been described in section 2.5.1.

Messaging The message plugin offers a message exchange service that allows for asynchronous inter-agent and agent-plugin communication. The Message plugin transfers simple text messages between sender and receiver and it is up to the communication partners to exchange meaningful information, e.g by using ACL-compliant messages.

Survival The survival plugin helps agents to reside on an agency in what can be call sleep-mode.

After an agent's execution is finished, the agent is stopped and destroyed and the thread which hosted the agent is returned to the tread pool. But there are many tasks an agent is faced with which require the agent to wait for a variable amount of time until some event occurs or a time frame has closed. For example, an agent may require a specific service that other agents can provide and therefore makes a call to all the agents. After making this call, the agent has to wait until it receives some answers or its time is running out and it has to continue its task.

To keep agents alive even after their current execution cycle has ended, the Survival plugin can be used. Agents simply request a context to this plugin before their execution ends. The Survival will prevent the deletion of this agent and can reactivate it at configurable times or intervals.

Shell The Shell plugin provides a simple command shell to administrate a running Tracy agency.

TAAS (Tracy Authentication And Authorization) The TAAS plugin provides sophisticated security mechanisms that can be adapted during runtime to allow for a fine-grained access control between agents and plugins.

Monitoring The Monitoring plugin provides statistical information about a running agency, e.g. the number and types of running agents and their current life cycle status like running, migrating, etc. An overview of installed and running plugins is also provided.

Furthermore, the Monitoring plugin provides an interface for the Tracy Administration GUI *Wai Lin*, which is described in detail in section 2.6.3.

NetMonitor The NetMonitor plugin allows for constant monitoring of a Tracy 2 agency network, e.g. the number of available agencies, latency and bandwidth between any two agencies. The collected information can be integrated into agent migration decisions.

NetworkManagement The NetworkManagement plugin provides a more sophisticated and robust logical network overlay based on the JXTA[15] peer-to-peer framework. It allows for the discovery and access of Tracy agencies and provided services in an otherwise unknown network. Compare figure 2.11.

Partitioner As mentioned above, the main research area of the Tracy team has been the optimization of an agent's migration process. Based on the flexible possibilities to configure agent migration behavior as provided by the Migration plugin, further optimizations have been evaluated [Kern et al., 2004; Kern and Braun, 2006; Braun and Kern, 2005].

One of them is available with the Partitioner plugin. The Migration plugin allows for the selection of single classes that should be transferred during a migration. However, many Tracy agents consist only of one or a few classes thus unnecessarily limiting the power of this feature. The Partitioner plugin is able to split agent classes into several smaller parts based on information gathered by static code and runtime analysis as well as heuristics. Thus, an agent's code is spread among a fine-grained set of small classes and a migration can be configured in a satisfiable and useful way.

[15]http://jxta.kenai.com

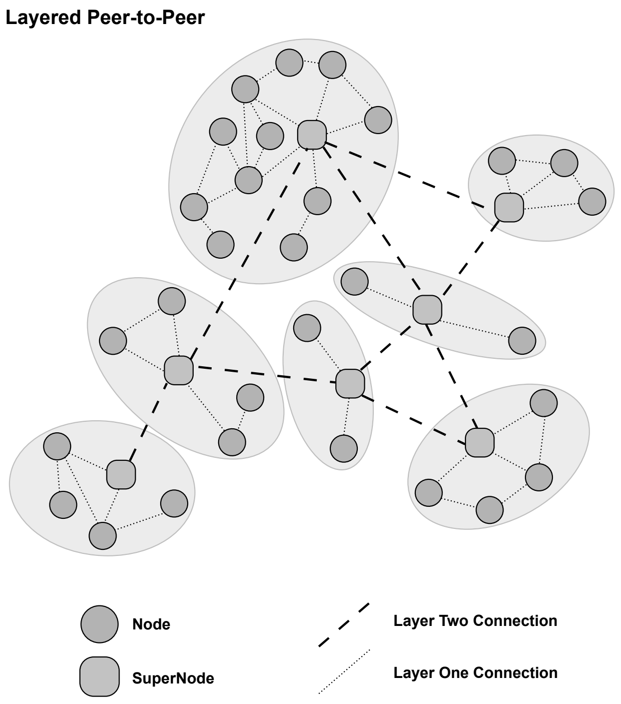


Figure 2.11: Network Management based on JXTA.

For more details about this process we refer to [Kern et al., 2004].

AgentTrace The AgentTrace plugin can be used to run automated migration tests in a network of agencies. The plugin is able to run several test cases in serial where each case may consist of any number of agents that perform a given number of tours among the available agencies. Each of those agencies must have the AgentTrace plugin running in order to gather statistic data like local agent execution times or point of time for in- and outbound migration. After running a test case, an agent will collect all the acquired data and return to the test running agency to create a comprehensive report.

AgentPersistency The AgentPersistence Plugin serves two main purposes. First, it enables agents to capture their current state to create a kind of checkpoint which can later be used to return to exactly that state. Second, it allows for capturing the state of all currently running agents and reestablish the whole agent environment at a later time. For example, this can be useful in case of a system restart due to a scheduled maintenance event.

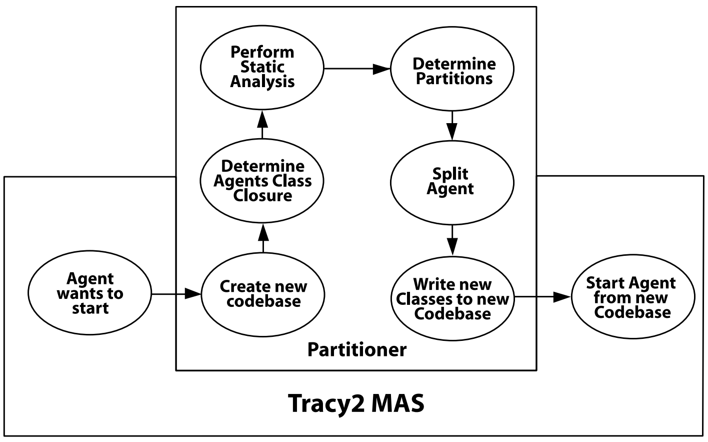


Figure 2.12: Process of Partitioning an Agent during Startup.

After discussing the general system architecture of Tracy and several of the more common plugins, the next section will take a closer look at Tracy's agent model.

2.6.2 The Tracy Agent Model and its Shortcomings

The Tracy Agent System features a very lightweight agent model which provides only the absolute necessary foundations for mobile agents to achieve a maximum of interoperability.

During the time when the conceptual work and initial prototype development have been conducted many different mobile agents systems have been in the wild. All of them claimed to run in heterogeneous environments, to interact and exchange agents with other systems. Whereas many of them were able to deliver on the first two claims, most of them failed concerning the third one. Which was, and still is, the most challenging problem. But mobile agents have been considered as a paradigm, which is capable to overcome the difficulties of a heterogenous environment, and therefore agent migration between and execution of agents on different platforms is absolutely necessary.

One vision for the second version of Tracy has been to provide a lightweight agent model that allows for agents, which can at least be executed on nearly every agent system that was available at this time. Due to the fact that the language of choice for mobile agent systems has been, and still is, Java, the Tracy team aimed for the smallest set of properties that all Java-based mobile agents had in common. As it turned out, this set was rather small. The only similarity between all mobile agents was their ability to be executed as a Java thread, e.g. they all featured a *run* method, and could be serialized. Thus, the Tracy team decided to abandon any concepts of providing a base agent class for Tracy agents, which could have delivered various frequently used functionality like agent state handling or a sound way of handling aggregated data. Instead, they made the existence of a simple *run* method and the ability to be serializable the only conditions to qualify as a Tracy mobile agent. In following this trail, Tracy agents can be transferred and executed between any Java-based agent system. In 2004, Braun et al. showed that the integration of the Tracy migration engine into the Jade [Bellifemine et al., 2007] system allowed for agent migration between Tracy and Jade agencies. Furthermore, it seemed to lower the entry hurdle for writing mobile agents to manageable level for even novice Java programmers due to the fact that it does not require the understanding of a complex agent model.

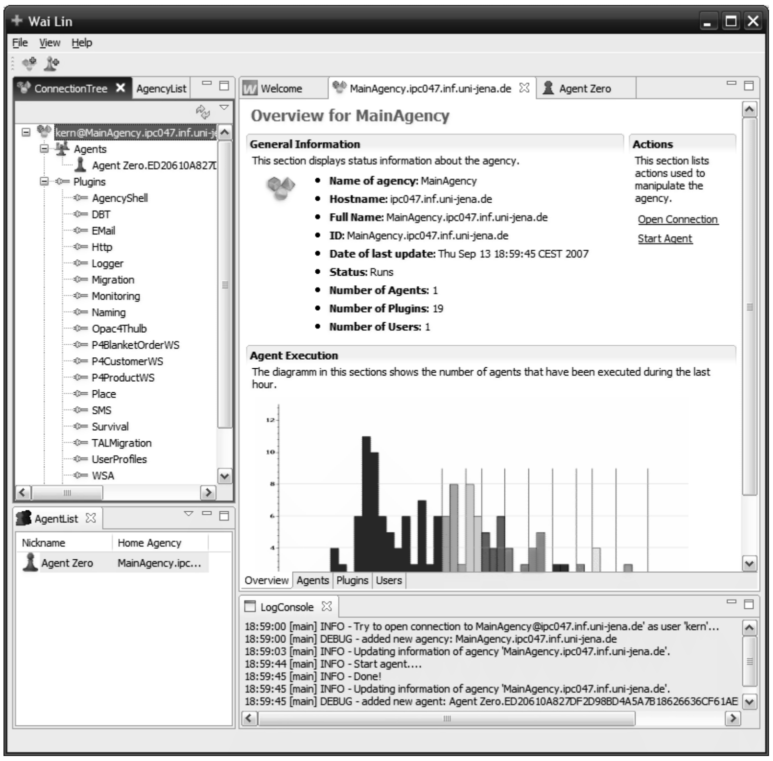
As consistent as this may seem, it did come – as we now know – with a large set of concessions. But at first, it seemed like a great idea, having no limitations on how to write a mobile agent. Each programmer could try out different ideas on what she thought would by an ideal inner structure for an agent. However, most Tracy agent programmers settled with a simple state machine that, depending on the complexity of the agent, resulted in a confusingly long list of state constants and conditional statements. These state machines were hard to understand and maintain. Moreover, most agent programmers tended to copy such an already written state machine to every new agent hoping to save time. While the time saving idea was always a false friend, this habit led to the existence of countless agents sharing nearly the same code structure but doing entirely different tasks.

Beside that, the non-existence of any basic foundation for an agent has led to various concepts on how an agent should use Tracy plugins. Tracy plugins are extensions to the Tracy micro kernel that provide additional functionality to the system. Plugins can be installed, started and stopped during runtime and are accessed via a dynamically created context. An agent needs to obtain a context object, which will act as a proxy between the agent and the desired plugin. Some of the basic plugins described above, like message exchange or migration, are needed by nearly every agent, but the code to use these plugins was newly written – or copied – for each new agent. The resulting problems are essentially the same as with the aforementioned state machine.

In conjunction with several shortcomings of the system as a whole, the lack of any structure and guidelines for agent programming also lead to a reduced value of Tracy 2 for teaching purposes. During their first steps like initial installation or getting an agent migrate between two agencies, students stumbled far to often to raise a greater interest in agent technology.

2.6.3 Tracy Administration

During the early stages of the MobiSoft project [Erfurth et al., 2008], it became apparent that the command line provided by the Shell plugin isn't suitable for larger application scenarios and doesn't appeal to *normal* users. Thus, combined with the information already gathered by the Monitoring plugin, the Tracy team decided to create a graphical user interface to administrate a set of Tracy agencies. The new frontend was named *Wai Lin*, after a fictional female, chinese secret service agent starring in the James Bond film *Tomorrow never dies*[16].



Nickname	Home Agency
Agent Zero	MainAgency.ipc...

Overview for MainAgency

General Information

This section displays status information about the agency.

- **Name of agency:** MainAgency
- **Hostname:** ipc047.inf.uni-jena.de
- **Full Name:** MainAgency.ipc047.inf.uni-jena.de
- **ID:** MainAgency.ipc047.inf.uni-jena.de
- **Date of last update:** Thu Sep 13 18:59:45 CEST 2007
- **Status:** Runs
- **Number of Agents:** 1
- **Number of Plugins:** 19
- **Number of Users:** 1

Actions

This section lists actions used to manipulate the agency.

Open Connection

Start Agent

Agent Execution

The diagramm in this sections shows the number of agents that have been executed during the last hour.

```
18:59:00 [main] INFO - Try to open connection to MainAgency@ipc047.inf.uni-jena.de' as user 'kern'...
18:59:00 [main] DEBUG - added new agency: MainAgency.ipc047.inf.uni-jena.de
18:59:03 [main] INFO - Updating information of agency 'MainAgency.ipc047.inf.uni-jena.de'.
18:59:44 [main] INFO - Start agent....
18:59:45 [main] INFO - Done!
18:59:45 [main] INFO - Updating information of agency 'MainAgency.ipc047.inf.uni-jena.de'.
18:59:45 [main] DEBUG - added new agent: Agent Zero.ED20610A827DF2D98BD4A5A7B18626636CF61AE
```

Figure 2.13: Wai Lin - Tracy 2 Administration Workbench.

The Eclipse Rich Client Programming Framework was chosen as the core for the new plugin as it provides a widely distributed and and well-known, easy to use founda-

[16]http://www.imdb.com/title/tt0120347/

tion. The plugin's structure is similar to common database administration tools. It offers means to connect to different agencies at the same time, access all the statistical information about plugins, agents and users. Furthermore, plugins and agents can be started or stopped. One especially nice feature is tracking of an agents tour among the connected agencies. In detail information about single migration and roundtrip times, delays and residence durations can be accessed thus allowing for a great system overview and agent debugging. See figure 2.13.

To access all these information, the Monitoring plugin has been extended to provide all its gathered information as a web service and to allow for the connection of clients, for example the newly developed graphical Tracy frontend.

Chapter 3

Thesis And Structure of this Work

So far, we have outlined the current state of research along with the drawbacks of our own agent programming model. We will now discuss the shortcomings of the most prominent agent model – BDI – and present ideas, aims and reasons for our own approach.

3.1 Drawbacks of BDI

At this point, we shortly recall the BDI architecture that was in detail described in Section 2.4.1.2 and analyze several drawbacks of this agent model.

Core elements of the BDI model are *beliefs*, *desires* and *intentions*. Beliefs describe an agent's knowledge about its environment and itself, desires mark an agent's goals and intentions are commitments to achieve one or more goals at a specific time. During each execution cycle, a BDI agent perceives changes in its environment and updates its belief base. Based on these new beliefs, the desires are updated, e.g. it could turn out that a single desire can no longer be achieved so the agent will drop it completely. Afterwards, the set of currently hold intentions is updated and one of these intentions is selected to be followed in this execution cycle. Intentions are usually mapped to predefined plans which are executed to work towards the achievement of this intention and thus the related desire. BDI systems have proven to work well in the field of multi-agent systems that are applied in closed scenarios, for example air traffic control.

At this point, one could suggest we should simply select one of the available BDI or reasoning architectures and integrate them into our agent system. However, there are several aspects that discourage a simple adaptation of a BDI system, respectively the BDI framework:

BDI systems disregard mobility performance As stated in section 2.5.3, current BDI systems are nearly solely focused on just the BDI part and the basic properties of agenthood. Many of them completely ignore mobility as a possible agent property while others allow for mobile agents but use rather simple implementations for migration. The reasons may be that most of these systems evolved in the traditional agent/multi-agent community. However, we start from the opposite direction with mobile agents and a mobile agent system trying to move towards this basic agent scheme.

BDI systems are barely suitable for typical mobile agent tasks As an example for a industry-proven BDI system, we refer to JACK [Padgham and Winikoff, 2004]. To achieve a performance that is suitable for real world applications, the JACK implementation disregards several aspects that make up a BDI system in theory. These are ad-hoc planning and adaptation of plans; JACK merely uses predefined plans that are just executed [Padgham and Winikoff, 2004]. Our aim is not to implement just another BDI system, thereby dropping important aspects of this model to make it applicable in real world scenarios. We rather want to learn from the conducted research and achieved results and construct a system that is applicable, usable, still flexible and maybe a bit *BDI-like*.

State-based/BDI systems are hard to understand and use This claim arises from the fact that, in most BDI systems, agents and plans are described in a kind of first order logic language. Despite the fact that logic should be common to every computer scientist and engineer, most of them do not use them daily or write programs in such a language. Thus, introducing a system based on a logic language into industry would at least be difficult. System architects and engineers would have to learn a new programming style, learn to build programs with it and acquire a base knowledge on this new kind of system. We believe that it would be much easier if such a system is written in a well-know and widely used programming language. Thus, one is not bothered with learning a complete new style of system development, but can concentrate on the aspects of the new paradigm by leveraging existing knowledge.

BDI systems have a fixed set of plans As mentioned before, most BDI systems use a set of plans, that are applicable in certain situations respectively usable to follow specific intentions. Thus, a pre-defined plan is mapped to an intention at runtime and is afterwards executed. Such plans are usually a set of activities which are structured in a certain order. Agents will receive a number of plans at start up and are meant to acquire new plans from other agents by means of interaction, collaboration, and negotiation [Ancona and Mascardi, 2004; Ancona et al., 2004; Jonker and Robu, 2004]. However, due to the fact that all plans must be related to some beliefs that make up an agent's world model, it seems to be quite hard to integrate foreign plans into an agent's world model at runtime. Recently, the Jadex team extended their system with a planning engine that is used in situations where no applicable plans can be found in the plan library. Given a hard time limit, the planner tries to construct a plan that is applicable in the current situation. If no complete plan can be derived inside the time frame, the best one found so far will be used [Walczak et al., 2007].

Balance between Reasoning and Acting The most critical aspect of a BDI system is to find a good balance between reconsideration and goal-directed behavior. On the one hand, a BDI agent should often enough reconsider its current behavior and reason about future steps to uphold its adaptability. On the other hand, if it does this too often, the agent will be completely occupied with this reasoning process and will not achieve anything because of ever-changing desires and intentions. Evaluations could show that seldom reconsideration works well in slowly changing environments whereas highly dynamic environments demand a more frequently reasoning [Fasli, 2007].

3.2 Towards Modular, Plan-based Mobile Agents

In this section, we will outline the system and architecture that we have in mind. However, before going into detail, we need to define several terms that are essential to the understanding of the upcoming paragraphs.

Task A task denotes user given goals that should be reached by the execution of an appropriate plan.

Plan A plan is a combination of actions that form a step-by-step guideline for an agent to solve a user-given task.

Action Atomic actions are the core building blocks of plans and, thus, agents. Each action is related to an activity or function that can be performed by an agent, e.g. send a message, migrate to a remote agency or query a directory service.

After introducing these terms we can now start with the description of our ideas. Instead of adapting a BDI agent model, where agents select plans and actions during runtime for execution, we aim at providing agents with a complete, fault-tolerant script. As such, the script is made up of one or more tasks and can be considered as the logical evolution of our old state machine model, only without the various drawbacks. First, handling of such a script should be as simple as possible to allow for easy creation and maintenance. Second, a task description should provide mechanisms for flow control, e.g. decisions and loops, and to handle runtime errors, e.g. the failure of a single action or a complete plan. At last, agents should be able to execute any script that conforms to a general model. Thus it would be possible to exchange scripts during runtime.

Further on, we will base our research on a real world scenario to keep track of the important problems. Many agent models like BDI have proven to work well with rather artificial examples like *Blocks World* [Bordini et al., 2009] or simple auctions [Bordini and Hübner, 2004]. Even in such easy scenarios, current BDI agent descriptions tend to be rather complex and hard to understand.

The main goal of our work is to ease the development process of mobile agents by introducing a framework that allows for the creation of agents based on a set of high-level building blocks. We aim at using the best of both worlds, e.g. the plan concept of BDI, which abstracts from atomic actions an agent can perform and the straight forward approach by programming agents in a widely used programming language, respectively environment, that is well-known to the typical application developer. Domain-specific plans that can be combined to larger scripts fulfilling more extensive tasks will help us describe reoccurring tasks in a generic, reusable way. Beside that, they abstract from the low-level programming of state machines as well as the configuration of migration steps, thus making the development cycle of mobile agents easier and faster. Furthermore, scripts may contain parts that can be achieved in any order or in parallel. Having the freedom to choose between these options, agents may exhibit a more proactive behavior than agents that are bound to a fixed sequence of low-level statements.

3.3 A Unified Approach to the Development and Usage of Mobile Agents

In the last sections as well as in chapter 2, we have described the drawbacks of current planning architectures for agents as well as the shortcomings of our own agent programming model. Further on, we have outlined our ideas concerning a new model that is easier to use for both, developers and end users, and that should satisfy current needs of applications based on mobile agents. Beside that, our ideas presented so far still rely on developers that *design/describe* an agent's behavior in a rather fine-grained way. With regard to the common understanding of an agent as an entity that works completely autonomous in an unknown environment, our proposals are far away from that optimum. But, taking a look at other current agent systems, non of them is able to live up to that vision. Some of them offer hard-wired solutions like our own Tracy 2 agent model whereas others have agents that are able to reason and plan – but only in closed, well-described environments. Furthermore, most of these systems incorporate several different languages to describe agents, making it a challenging task to implement advanced behavior.

Regardless of the appealing vision of an autonomous agent, we are not sure if such a high degree of autonomy is desirable or useful. There might be many cases where an agent that follows a strict, given execution script will outperform such a *free* agent. Taking this for granted, a flexible architecture that allows for agents to exhibit both types of behavior – and everything in between – seems to be a promising step. For example, when time is critical, an agent may skip planning completely and follow the trail given by the plan. Contrary, if the agent is waiting for some events or a service, it may start to reason about its current state and decide on its next steps. What we aim for is to provide a sound, architectural basis for agents that will allow for the creation of agents that use fixed plans as described in the last section.

3.4 Theses

In summary, we aim at reinforcing the following theses:

Thesis 1: It is possible to develop an agent model and a runtime environment to execute predefined plans by using a well known, industry proven programming language like Java

To allow for an easy and fast adaptation of a new agent model respectively agents in general, it is highly desirable to use well known and widely used tools and programming languages. Thus, we aim at creating an agent model and execution engine using the Java programming language.

Thesis 2: It is possible to create an agent development environment that is easier to use than any of the currently available ones

Regarding the fact that sophisticated concepts for modularizing on the intra-agent level are missing from most software agent frameworks and methodologies [Pokahr et al., 2010], our goal is to separate the creation of atomic functionality, e.g. actions, and the definition of execution scripts based on such actions. Whereas action creation should be performed by programmers using the Java programming language, we clearly target non-experts as the primary user group for the definition of agent scripts. Thus, making the model as well as the tools as simple as possible is one of our aims.

Thesis 3: The proposed framework will allow for faster development and execution cycles as well as provide better software quality

With an easy to use toolset that allows for the creation of agents out of existing atomic actions, the time required to create new agents or adapt existing ones should be much smaller than the efforts needed to perform the same task at code level. Furthermore, with small, single purpose actions as building blocks for agents, the overall software quality in terms of maintainablity, reusablity and general code quality should be increased.

Figure 3.1 visualizes how this work should fit into the landscape of available agent systems and their approachability by different user groups. Whereas nearly all current agent systems and development frameworks target the agent expert group, we clearly aim at the two other user groups that far less familiar with this technology. Additionally, we intend to use standard tools and frameworks to create our new agent framework.

3.5 Structure of this Work

After having defined the goals for this thesis, we will outline the steps we aim to take to achieve them. Please compare with Figure 3.2, which depicts these steps and their ordering. In the following sections, we will outline each of them in detail.

3.5.1 Agent and Task Model

At first, we will have to create an agent model that describes a mobile agent in an abstract, platform and programming language independent way. This includes how an agent is internally structured, its set of generic, interchangeable capabilities, and so on.

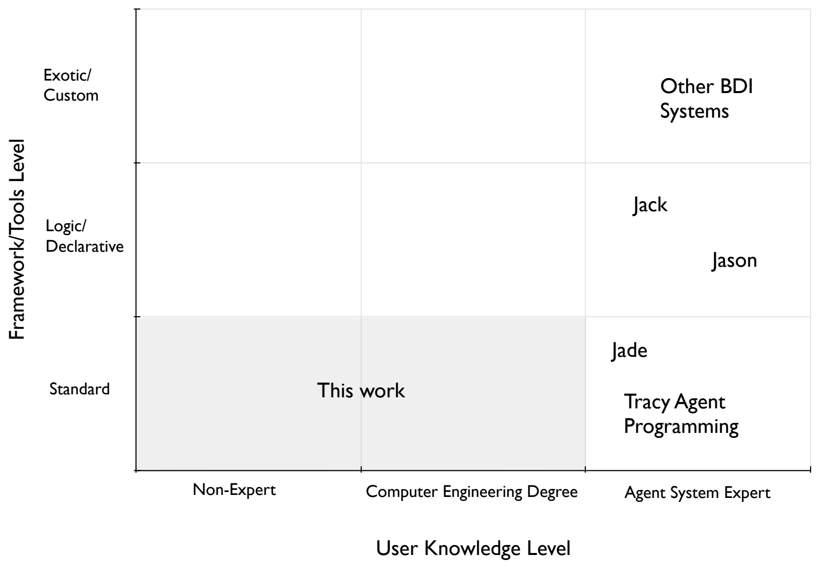


Figure 3.1: System Classification

Second, with the abstract agent model at hand, we can start to work on an abstract task and plan model that allows for describing execution plans for our agents. With this model, one should be able to describe typical tasks for a mobile agent like information retrieval, network node maintenance or observation tasks. We currently perceive a Petri Net like separation of states and transitions as a promising way. Transitions are the actual actions performed by the agent whereas states will describe distinct agent states and control the flow of actions by introducing transition selectors that allow for loops, parallelism and error handlers.

3.5.2 Model Implementation and Integration into the Tracy 2 Mobile Agent Toolkit

The second step of this thesis is concerned with the implementation of the proposed agent model in a specific mobile agent system. We will use the Tracy 2 toolkit, presented in section 2.6, for this purpose.

3.5.3 Implementation of a CASE Tool Prototype for Agent Creation

Due to the fact that we aim at easing the development effort for mobile agents, we will implement a prototype of a graphical design tool that should allow for creating agent models and plans with a rich and intuitive GUI. This application should abstract from programming languages and the mobile agent toolkit as far as possible and support the combination of atomic, self-contained actions into complete plans and scripts.

3.5.4 Evaluation

The last part of this thesis will be the evaluation of this new development and execution framework and a comparison with the current state of the art. We aim at showing that our new framework is equally powerful and fast while at the same time decreasing the development and maintenance effort for mobile agents, as well as reducing the number of errors produced.

While performance issues can be compared after several tests, comparison of the development effort is more complicated as this is a more subjective parameter. We intend to let students program agents, some of them using the new framework while others use the current agent model. Furthermore, we will check the approachability and ease of use of the created tools with an evaluation that is targeted at non-experts.

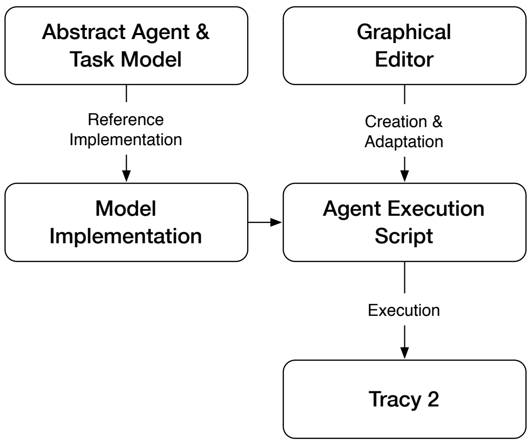


Figure 3.2: Outline of Thesis Steps

3.6 Assumptions

For this thesis, we make the following assumptions, respectively restrictions.

First, we are not focused on AI techniques to improve agent design and behavior. Our main intent is to ease the implementation effort for mobile agents which usually have rather straightforward, standalone tasks. Some of the possible extensions to this work, like automated plan creation or dynamic decision for a network communication paradigm, e.g. remote access or migration, may have a higher impact from AI. However, these topics are not our main concern.

Second, we are not concerned with semantic service descriptions, dynamic service orchestration, respectively composition or semantic matching of requests and offered services. At those points, where our agents are meant to access a service, we will assume that an appropriate service or a list of services can be found using straightforward techniques like a directory service.

Third, we will not be able to provide a final stable implementation of the agent development tools that allow for a creation of agents based on plans and actions. We will merely implement a functional prototype that is able to show the general concepts and strength of our approach.

Further on, we do not see our work as a direct competition to BDI-based approaches. With BDI, the actions that an agent will perform are selected dynamically at runtime. In our approach, we aim at providing a sound, optimal plan for a complete task in advance. With extensions like automated plan creation or dynamic insertion, exchange or deletion of actions, we would move a bit more towards a BDI-like approach. However, in any case, we do not apply mental concepts like believes or desires.

Chapter 4

Problem Domain, Scenario and Roles

In this chapter, we will outline the problem domain that will be covered by this thesis. At first, we will describe this domain in a general way followed by a specific application scenario that should provide a more demonstrative picture. We go along with the presentation of participant roles, which we aim to introduce into the development process and usage of mobile agents. Afterwards, the scenario and the problem domain will be revisited by taking into account the newly introduced roles.

4.1 Problem Domain

The first ideas to this work arose during our work on migration optimization on code level. Our attempt was to reduce the size of the transferred code as much as possible by means of class splitting [Kern et al., 2004]. Further on, we implemented a simple migration planner that was able to calculate an optimal sequence of migration steps for a given tour and environment parameters [Kern and Braun, 2006]. During this research, we realized how inflexible and programmer-dependent our mobile agents were. With respect to a maximal compatibility to other agent systems, Tracy 2 agents are simple Java classes that implement the *Serializable* and *Runnable* interfaces. Thus, the only assumption one can make with respect to a Tracy 2 agent is that it contains a *run* method. Further on, a programmer cannot rely on functionality that is provided by a base class, which means that with every new agent one starts from scratch.

A typical Tracy 2 agent is based on a state machine, that is used to model the agent's task and its behavior. Actions that are performed by the agent, e.g. a migration or sending a message, usually lead to a state change. Depending on the scenario, such state machines tend to become very complex and implementation soon gets tedious and error-prone. Moreover, later changes or extensions become hard to nearly impossible.

What makes this state machine based architecture even worse is that the agent is bound to its initial program, e.g. the state machine, which is given by the programmer. During runtime, the agent acts according to the state transitions and will either succeed or fail. There is no way to alter this state machine during runtime, for example by exchanging some parts, or to introduce a completely new one. Nor can the agent adapt to changes in the environment that have not been anticipated by the programmer in advance.

4.2 Scenario

Research is often conducted without a concrete aim just investigating an interesting idea by working in some direction and performing some experiments which in the end would hopefully lead to interesting results or at least a specific research project. For many researchers, this surely is the most interesting way to work; to have no time or topic constraints while moving freely across different research areas. Unfortunately, one may get lost in the depths of science without achieving anything. Thus, this thesis will be conducted with a specific development and application scenario in mind. This scenario should help us to evaluate the achieved results, keep us focused and provide a stable base for discussions. We have chosen a well-known scenario, that of a traveling researcher, for basically two reasons. First, this scenario is common to most people, so going into productive dialogs with other researcher should be easy. Second, the scenario has been widely discussed in literature, so we can learn from these experiences and compare our results.

Consider the following situation. A researcher has successfully submitted a paper to a conference and is now planning his attendance. This includes conference registration as well as booking a hotel and organizing the journey. As usual, the researcher has to keep an eye on several parameters like travel budget, distances, timing, and personal preferences that can influence one or more of the three subtasks. For example, having a tight budget, conference attendance may only be possible if the researcher can get the reduced early registration fee. If not, the whole travel must be canceled.

In the following, we outline a possible flow of actions from the scientist's point of view. At first, the need for organizing a travel arises and, ideally, the scientist's personal agent would automatically derive an adequate task by combining information from various sources like its owner's calendar or mailbox. However, as such a step includes adequate semantic descriptions of information resources and advanced reasoning, it is far more likely that the scientist himself assigns this tasks to the agent. This step may be performed using a desktop PC, a Tablet or even a mobile phone. Thus, it is desirable, that such a task delegation is as intuitive and easy as possible. After having received the task, the agent should be able to work on its own, thereby moving around the network in search for necessary services and information. The agent pursues its task until one of three cases occurs: it has either finished its task, the task requires a callback or some error arose. The first case is the most preferable one as it frees the scientist from any further actions, but such a case is likely to occur only in the simplest information retrieval tasks. In the second case, the agent may have acquired enough information to present its owner a set of alternatives, for example, in our scenario a number of hotels that all fit equally well into the given parameters. Here, the agent cannot respectively should not make a decision on its own, so it will let its owner decide. The third case, the error case, is not desired but must be taken into account as errors and failures are likely to occur in an open scenario like the one under consideration. For example, the agent may not be able to find an appropriate service that delivers hotel information. In such a case, it should try to solve the other subtasks and present these preliminary results to the user, who would decide on the next steps. Finally, if the agent has successfully solved the given task, it would return to the scientist and deliver the results. The scientist could now control the results and, in case of an indisposition, charge the agent to provide a refinement or correction. At last, if the results are as expected, the task is considered to be solved completely.

4.3 Roles

At this point, we will more clearly specify three roles and describe the way in which we aim to use them throughout this work. First, by *developer*, we mean someone writing code, e.g. the code that implements an action, without knowing much about the context or application in which this piece of code will be used. Second, a *designer* works on a higher level by actually designing agents and their behavior. The designer will combine single actions into a bigger plan that is executed by the agent. Last, the *end user* is the person who will actually use the created agents, i.e. the scientist in our scenario.

4.3.1 Developer

By developer, we mean programmers, who actually implement agent functionality using a widely used programming language like Java. Our aim is that the developer is no longer in charge of programming complete agents for each complex task. Instead, we intend to let developers provide a set of core actions that can be performed by each agent and that are highly generic and can be combined to complex tasks. Such *plans* should be exchangeable during runtime meaning that an agent becomes a simple execution engine for plans. Low-level actions are, for example, sending a message to another agent, initiating a migration or query a service directory. Low-level actions should be very generic and self-contained so that they can be combined in all possible and reasonable ways. Actions should conform to a standard interface and provide input and output sets that can be mapped to adjoining actions.

4.3.2 Designer

By looking at our problem and scenario description and the aforementioned roles, there is an obvious gap between the low-level implementation work conducted by the developer and the usage of plan-based agents by the end user. To close this gap, we introduce the *designer*. A designer is in charge of creating high-level plans that solve complex tasks by combining low-level actions provided by a developer. The relationship between developer and designer must not be strictly one-way. If a designer encounters the need for a new action, then a developer can be charged to implement that functionality. Moreover, our ultimate aim is to merge the end user and designer roles. Using agents would be more flexible, if an end user could not only select one of several plans for a given task, but would be able to create new plans on its own. This process of creating new plans could be supported by the application in various ways, i.e. by suggesting actions that would make sense at a specific point in the edited plan. To further ease the plan creation, we could hide the low-level actions from end users by introducing *macros*: collections of actions that deliver a solution for smaller tasks respectively commonly used subtasks.

4.3.3 End User

Compared to our scenario, the scientist is an *end user*. As the name implies, this role describes all people, who will use the final system respectively agents. From their viewpoint, delegating a task to an agent should be as simple as possible. In our travel example, the scientist should just have to submit the destination and a time frame to its agent. It would even be better, if the agent could derive the task by reasoning about information it has about the user, for example entries in a calendar or emails. After having acquired the task, the agent should solve it with as few user interactions as possible. There may be incidents where the agent is not able to make a decision and has to ask its owner. For example, if a condition given by the end user cannot be guaranteed or an error has occurred. However, the perfect case would be that the agent does not bother its owner at all and just delivers the final trip plan.

4.4 Scenario Revisited

With the roles defined in the last section, we will now come back to the scenario to review it under this new perspective. In short, we intend to delegate the task for organizing a conference attendance (or a travel in general) to a mobile software agent, that will try to solve that task on its own by using a high-level plan. That plan was created by a designer in advance using basic generic actions that can be performed by each agent.

Before our scientist is able to assign the travel booking task to its agent, several steps must have been performed by one or more developers respectively designers. Given that the base system with an agent and plan model already exists, the developers would have to implement actions that enable an agent to use services that provide for example hotel information and reservation/booking functionality. Here, we think of atomic actions that provide access to a service and other actions that are able to evaluate the results of a previous service access according to given parameters.

These atomic actions would be weaved together by a designer to create a high-level plan for arranging a travel. Such a plan could consist of three subtasks, namely *Book a Hotel*, *Arrange Travel* and *Register at Conference*. Each of those subtasks may be available as a macro and specifies its own properties which should be achieved by the solution, e.g. price range, maximum distance between hotel and conference or means of travel. Global properties that belong to the high-level plan and which affect all subtasks are, for example, time and location of the conference and minimal overall price. The designer has to take care that the final plan is robust, complete and fulfills its aim. Beside that, the plan creation process should be performed in close collaboration with the developers to satisfy the need for new required actions or to adapt existing ones. This tight feedback cycle should help to improve the plan model implementation and increase system reusability and robustness as well as improve and extend the set of available actions.

The scientist would, as in the scenario description at the beginning of this chapter, assign the travel booking task to its agent. The agent would be configured with the appropriate plan and the parameters specified by the scientist. Afterwards, the agent would start to execute the plan, thereby solving the task step by step as intended by the designer. From the end users point of view, everything works as in the first scenario description. Either the agent comes back with the results of the solved task, with a request or to report an error case.

A second addition is concerned with preparing the conference attendance. When arriving at a conference, most of the other attendees will be strangers to the scientist. At the end of the conference, he will have heard many talks and knows who works on which topics. Now he could start interesting discussions with people working in the same area as he does. But the conference is over and everybody is on its way back home. So it would be nice if he had some of these information at the beginning of the conference. Knowing who is doing similar research would help to find interesting colleagues right from the start. So this could be a task for an agent; gathering information about other conference attendees and filtering out those which are in some way related to its owners research area. The agent could even collect papers published by those scientist which the scientist can read during his travel. Thus, he would not only know who might be interesting, but would also be well prepared for a lively discussion.

4.5 Problem Domain Revisited

To develop such application as described above with our current Tracy 2 state machine based model, one would end up with a very complicated state machine containing an awful number of states and many ambiguous interconnections. First, the developer would have to model the agent's different tasks thereby anticipating possible migrations during the execution of a single task. Further on, a great number of states and interconnection would be necessary out of robustness reasons, i.e. to recover from failed migrations or handle cases where a service is unavailable.

A slightly better solution would be the introduction of several state machines; each one covering a single task that can be executed on its own. Such a solution seems to be possible with subtasks that do not intersect. However, in usual application scenarios, subtasks are not autonomous but depend on each other in one or more ways. For example, in our scenario, the gross price of a conference attendance may have an upper limit, so the sum of costs for travel, hotel and registration should be less or equal to this given bound. So, when having related subtasks, one would have to connect the different state machines, which brings us back to the original huge one. Or, when applying communicating state machines [Brand and Zafiropulo, 1983], we end up with a huge set of interconnected state machines that are difficult to understand and maintain.

Based on these facts, we intend to create an architecture that makes Tracy 2 agents more easy to use, flexible and error-resistant. One could say that we want to move from our state machine based agent model towards a state-based architecture as described in Section 2.4.1.2. Here, it seems reasonable to stress the differences between these two architectures. The first one, our current state machine architecture, is rather simple – agents are controlled by a state machine with fixed state transitions. In the latter one, the state-based architecture, an agent explicitly maintains a state, which has links to previous, incoming transitions and a number of outbound transitions to possible future states. Beside that, the state holds information about the environment and the current internal status of the agent. Thus, the agent has much more information to decide on its upcoming steps. Beside having our agents use a state-based model, we would like them to better satisfy the properties of software agents given in section 2.4. They should be able to adapt to changes in the environment and alter their flow of execution in order to achieve their given goals.

Our defined roles and their corresponding scope of duties allow for a clean separation of concerns and introduce modularity of application components that reduce coupling and increase reuse. These roles already render the outline of the approach we aim to take. We argue for a separation of concerns similar to that used in web application development today. First, there are *real* programmers who implement business logic in an industry accepted and widely used programming language like Java. Second, there are designers, who create an appealing user interface on top of the business logic without knowing much about the underlying implementation. They use their own languages like HTML (Hypertext Markup Language), JavaScript or a combination of these like Ajax.

In our case, we would like to separate the development of plans that fulfill a specific task from the creation of agents which are mere execution entities for such plans. Developers simply implement business functionality and each of those core plan elements is a single component with a small interface and several constraints, that describe the context in which this element may be used. In contrast, designers would model the overall use case by combining these elements into high-level plans. Ideally, they would use a graphical editor that allows for an easy creation of those plans by aiding the designer in a variety of ways. For example by providing a list of applicable actions at a certain point or showing the violation of a constraint.

In the first place, our idea may sound similar to recent service oriented approaches: applications, that use various services and that orchestrate those services into more complex workflows. However, there are several differences. Our plans aim at a higher level of abstraction which in the end should even enable end users to create plans for agents. Our plan elements will partly consist of elements that access services or activate complete workflows. But these element should abstract from the underlying technical details. Especially, these details are the main concerns in SOA research communities.

At this point, one could argue that we may end up with the same problems as with the state machine – having a single, fixed plan seems to be equally inflexible as a fixed state machine. However, introducing sophisticated error handling techniques into a modular plan is a relatively straightforward task. Far more convincing is the fact that creation and maintenance of plans is much easier than programming state machines from scratch. As said above, we even aim at merging the roles of end user and designer to allow for a more flexible usage of agents. But this depends on a very intuitive and powerful plan model and our work is clearly focused in that direction.

Chapter 5

Specification of the TAMo Model

5.1 Introduction

The aim of this chapter is the introduction of the general concepts and aims of the new modeling language for the Tracy 2 agent toolkit. We will call it *Tracy Agent Model* or shortly *TAMo* (pronounced like the italian *ti amo*). Beside the general overview, a comprehensive description of its elements will be given. Due to the fact that TAMo evolved in two iterations, the chapter's structure reflects this evolution.

5.2 Aims

In this section, the main goals for the development of a new modeling language for mobile agents are outlined. Despite the fact that there are a lot of modeling languages and methodologies for the description and creation of agent as well as agent based systems, we belief that all of them fall short when it comes to sheer simplicity. And in our opinion, simplicity of the available tools and toolkits is the sole lacking property of agent-focused development tools to allow for a wider adoption of the concept.

Specifically, all of the available tools require a profound knowledge of agent based systems and their special properties. The approachability of these tools for non-experts is very low and thus prevents a large audience from creating and using agents. Looking into history, one main goal of the agent community was to establish agents as personal assistants to normal users who could just use such agents to accomplish tedious or long-lasting tasks. However, research moved into another direction – in creating agent based systems that are able to model and solve highly complex tasks which involve a society of numerous agents that cooperate to achieve a higher goal. While this is a tremendous achievement, it simply neglects the normal user that is just in need of a personal assistant. Such an assistant should be able to solve the given task nearly on its own by accessing legacy systems such as a database or a web service. If necessary, a personal assistant should also be able to communicate with other agents, but its main focus remains to solve a single task for a single user, at least in our scenario.

5.2.1 Simplicity

The ultimate aim of TAMo is to offer an approachable and easy to use solution to the creation, adaptation and execution of mobile agents. Creating new agents should be a matter of minutes and not hours or days whereas altering an existing agent should not involve the examination of hundreds of lines of code. The tools should be coupled to the Tracy 2 agent toolkit to allow for a fast execution of created agents and thus offer good development turnaround times. Furthermore, simplicity of the whole concept as well as the provided tools is a mandatory requirement to make TAMo usable by non-experts.

5.2.2 Approachable by Non-Experts

The idea of a personal assistant that is able to solve specific tasks behind the scenes is very appealing, especially to non-experts respectively normal users. However, exactly this huge audience was not able to use such assistants because the available tools and frameworks required a degree in computer science to achieve any result at all. With its foremost aim to be as simple as possible, TAMo should be the first environment that allows for the usage of mobile agents by non-experts.

5.2.3 Using only Common, Well-known Technologies

Among the available agent systems and toolkits, it is very common to leverage the power of rather unusual programming languages and frameworks. For example, BDI based agents are usually created using a logical or declarative programming language. Furthermore, several systems introduce custom languages to design agents or extensions to the core system. While these specialized methods provide powerful means for experts, they present a huge entry hurdle for any non-expert. Thus, we decided to create our agent development framework solely by using well known, industry accepted languages like Java and its offered APIs. Moreover, the creation of new functionality for an agent should require no, or just a minimal amount, of expert knowledge of agent systems. Thus, we believe to reach anyone that is familiar with the Java programming language.

5.2.4 Improve Software Quality

As mentioned before, the general software quality of Tracy 2 agents has been rather low. Due to the fact of no given base framework or reference structure, developers usually started to create new agents by recycling old ones and adapting them to the new needs. However, altering complex state machines of old agents is very error prone. Moreover, most developers settled with creating just a single Java class for an agent thus increasing the coupling and making it very hard to reuse parts of an agent.

By introducing a model that describes an agents tasks as a sequence of self-contained, atomic actions, we aim at achieving much better code quality and allow for easier reuse and maintainability.

5.3 Core Concepts

The main goals outlined in the last section lead to several requirements which TAMo should fulfill. Foremost, TAMo should be as simple as possible because we believe that simplicity is the foremost property required for a wider adoption of a new technology or concept. Other requirements include for example a high degree of reusability of existing parts, fast and easy combination of those parts and sophisticated means to cope with error cases. In this section, we will describe the general ideas and building blocks of the TAMo model as well as the development framework and tools. But before, we will take a look at similar available systems and stress the need for a simpler and more approachable model and toolkit.

5.3.1 Related Work and its Implications

In this section, we will introduce several systems and research efforts that are directed in a similar direction than our proposal. We will highlight the differences and explain, why these systems are not sufficient for our needs.

The first system is the *Jadex Active Components* [Pokahr and Braubach, 2009; Pokahr et al., 2010] middleware which provides a managed execution environment for active components. In contrast to traditional (passive) components, active components exhibit a certain amount of autonomy regarding their execution. Instead of just reacting to requests, they can actively decide to perform some actions. Originally, Jadex [Pokahr et al., 2005] started as an extension to the JADE agent system that provided support for cognitive BDI agents [Rao, 1996; Walczak et al., 2007]. With the evolution of the whole platform into an active component middleware, the agent part became an extension to the new system. A second extension is *Jadex Processes*[1] which provides execution facilities for BPMN- and GPMN-based workflows [Leymann and Roller, 2000]. Similar to our proposal, they provide a graphical editor to create workflows and execute them on the middleware. However, both extensions are independent of each other and the level of interaction between them remains unclear. Can workflows be used to model agent behavior or will workflows integrate agents as part of their execution?

Another system, that integrates workflows in an agent system is WADE [Caire et al., 2008b]. WADE extends the JADE agent system with the ability to execute workflows. Workflows can be created using an Eclipse-based graphical editor called WOLF [Caire et al., 2008a] and are afterwards exported to Java code. There are several kinds of actions that can be added to a workflow, e.g. to control the flow of execution or access remote services. Even a container action that can be filled with custom Java code is available. A single class file is created for every workflow and is executed by special workflow agents. Due to the fact that the mapping between workflow and code is rather complex and imposes various constraints, using the system requires profound knowledge. Thus, it is clearly targeted at experts and not at end users. Interesting to note, the authors claim that, with the mapping to Java classes, they have introduced inheritance to the workflow metaphor as one could use an existing workflow class and extent it to create a more specialized version. This inheritance is completely backed up by the Java language respectively virtual machine and, curiously, no information is given on how this mechanism is introduced to the whole workflow creation process. For example, how are both workflows, the original and the inheriting one, linked and represented in the editor and how are changes to the parent workflow propagated to child workflows?

Whereas both presented systems introduce the workflow metaphor into agent-based systems with an accompanied graphical editor to define such execution flows, there a several differences when compared to TAMo. For example, both systems are clearly targeted at agent experts and therefore posses a rather steep learning curve. Having non-experts create agents using these systems seems to be nearly impossible. Another aspect is the coupling between the script engine and the agent system. Whereas TAMo can be used as a standalone engine to execute any kinds of scripts, the systems presented above are highly coupled with the underlying agent toolkit. Furthermore, there is no

[1] http://jadex-processes.informatik.uni-hamburg.de

strict separation of code and graphical workflow definition. Both systems offer a kind of *container* action that can be filled with arbitrary code during the design process. From a bird eye view, this provides the same functionality as TAMo with its predefined actions. But it does, by no means, offer the same level of reusability for single actions that are part of a script. Considering the kind of transitions between atomic actions, both systems use two kinds of transitions, successful and failed, with conditions described in the graphical editor. We believe that our concept of actions defining the number and kind of outgoing connections thereby hiding the internal conditions that lead to one or the other connection is more flexible and easier to use and understand.

Ultimately, we aim for a lightweight model, respectively toolkit, with a minimal set of elements that is easy to understand and use but that provides means for aggregation to structure model instances and, thus, increase maintainability and reusablity of software agents. In the following sections, we will outline the core concepts behind the proposed model followed by a description of the implementation that was carried out in two iterations.

5.3.2 State Machine

As mentioned in section 2.6.2, nearly all agents that have been developed for the Tracy 2 agent toolkit where backed up by an internal state machine which captured the different execution states and, in case of mobile agent, locations of an agent during runtime. As simple and straightforward as this concept is, the approach becomes nearly unusable for any but the most simplest agents. Any moderately complex agent will feature an enormous amount of states with countless transition between them. Trying to handle such a state machine by hand without any tool support is extremely tedious and error prone.

However, due to its general simplicity regarding the core concepts, we decided to use a state machine as the starting point for the TAMo model. Moreover, in its first version, the TAMo model itself was very similar to a state machine with elements such as States and Transitions. While working with the first version of TAMo, it became apparent that this direct approach is to technical and complex. These insights led to the simplification of the TAMo model and the creation of the second version of the framework. While still being backed up by a state machine under the hood, the creation of plans for agent has been simplified. States and Transition are gone and an agent designer will only connect Action elements in a meaningful way. In section 5.5, we will outline the transition of the first to the second TAMo version in more detail.

5.3.3 General Execution Engine and an Extented Version for Tracy

At first, TAMo was clearly aimed as a framework to ease the development of agents for the Tracy 2 toolkit. However, during the adaptation that were made for the second version of TAMo, we decided to separate the core parts from those which are tightly coupled to the agent system. Thus, TAMo can be used as a standalone model and engine to create arbitrary execution flows as well as a framework to create agents for the Tracy 2 agent toolkit. The core model respectively framework can also serve as starting point for an integration of TAMo into other agent systems.

5.3.4 Actions

As mentioned several times before, TAMo is based around the idea of small, self-contained, atomic building blocks that, properly connected, make up the execution flow an agent. We are calling those building blocks *Actions*. Every action fulfills a single specific purpose like accessing a database or web service, sending a message to another agent or performing the migration to another agency. The requirements of an action to be added to an agent should be as low as possible, e.g. actions should by no means rely directly on other actions. Interchanging data between several actions is achieved by the usage of a shared associative memory space where every action can read and write. Beside a success and error connection, every action can furthermore define any number and kind of outgoing connections depending on the possible outcomes of this action.

5.3.5 Task and Plan Layers

Beside the atomic actions, we aim at providing several other layers of reusability. Therefore, we introduce the notation of Task and Plan. Tasks are high-level building blocks of an agent and are connected with each other in the required execution order, e.g. a Task that determines the possible travel locations should be executed before the Task which will select and book the best of those locations. Each Task is just a container for a number of Plans, whereas each Plan is able to solve the Task it is contained in. Using several Plans in a single Task increases robustness and error protection, as a Task can select and execute another plan, if the previous one fails. See figure 5.1 for an illustration of this concept.

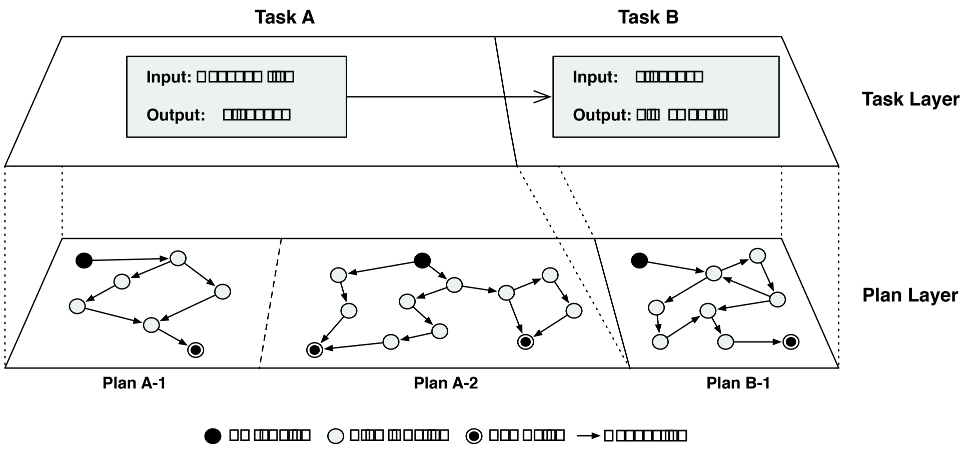


Figure 5.1: Task and Plan Layer

5.3.6 Separation of Development and Design

As described in chapter 4, we aim for providing a conceptual model that allows for the separation of development task between a core developer and an agent designer whereas the latter one doesn't need to write code but can use a given set of basic building blocks to create new agents.

5.4 First Version of TAMo

In this section, we will describe the first TAMo model starting with the main building blocks followed by the necessary *glue* elements. Thereafter, the presentation of the graphical notation for the different TAMo elements is given.

5.4.1 Model Elements

5.4.1.1 Main Elements

Script A script is the outer shell for everything that is relevant for the current objective. Every other TAMo element must be contained in a script or in an element therein. Every TAMo-based program respectively agent will be initialized with a single script that will be analyzed and executed by the engine.

Task To be of any use to an agent, a script needs to contain at least one task element. A single task is considered a set of plans to achieve one or more goals or obtain a specific result. For example, a task might be to access a set for RSS feeds and filter all items for some given keywords. The result would be a list of RSS items that match those given keywords. Another example could be to traverse a set of agencies and execute the same actions on each agency, like collecting performance measurements or installing a new plugin. A script may contain more then one task in which case these tasks can be structured hierarchically. This hierarchy describes the potential dependencies between tasks, e.g. to perform task B another task A must have been completed successfully. The separation of an agents objective into different, interchangeable tasks leads to a better, easier to understand script structure and better reuse of functionality by integrating already created tasks into new scripts.

Plan Every task element must contain at least one plan which describes a concrete flow of actions that will accomplish the single task. However, more than one plan can be present in a single task. All these plans should be able deliver the same results to achieve the task, but they may use different strategies, enact different services or use a different flow of actions. The introduction of multiple plans for a single task fulfills two purposes. First, robustness is increased significantly, if we allow for an agent to have alternatives to fulfill its tasks. Second, performance can be measured during various executions of the same task with different plans, so we can gradually increase an agent's performance over time by analyzing former executions. A set of plans adds flexibility to the system that would not be present, if a single task could only be achieved by a fixed, single set of actions. Despite the fact that we do not integrate AI techniques that would enable an agent to adapt a given plan in case of a failure, we believe that our approach provides enough robustness and flexibility and at the same maintains high execution performance and ease of use. Which plan is used when can be configured by the designer. So, for example, one can imagine that an agent should use different plans depending on various environment parameters like time of day, available network connections, time to complete the task, or desired quality of the results.

Action Actions are the core functional elements of TAMo. It is a single execution step towards the fulfillment of the enclosing task. An action may be as simple as writing a message to the agencies terminal but can be as complex as accessing a web service or a remote database follow by extensive filtering of the returned results. However, the simpler an action is, the higher is its reuse value. Moreover, more atomic actions generally offer a higher robustness and produce lesser errors.

State A state element captures a specific agent respectively world state during the execution of an agent workflow. State elements are linked with each other by executing a specific action which transforms the first state into the second. A state can hold any number of key-value-pairs that describe current agent properties or available resources. For example, after having acquired a list of data items from a web service by executing an action, the following state could hold a key-value-pair like *ws-data-items:=[List of Elements]* which depicts the successful execution. Other entries to think of are the current execution platform/host, a list of know hosts to migrate to, or some credentials to access specific services. The introduction of states into the model serves several purposes. First, by explicitly modeling an agent state at a specific point in a plan forces the workflow designer to think exhaustively about the current process and to explicitly capture relevant execution and agent state information. Second, having concrete state elements, we can check if the current agent state matches the ones present in a plan and, in case they don't match, deduct errors as well as create proper error messages. One could even think of introducing planning at a later stage of the project to build a new sub-plan in order to align the current agent state with the required state models in the overall plan.

Migration Despite the fact that, seen from a bird's view, a migration is a simple action, but we consider it as a first class element of the TAMo model as it is a vital part to any mobile agent system. And, seen from a closer perspective, it is much more complex than a normal action. The agent itself will initiate the migration and specify the destination as well as the used migration strategy.

5.4.1.2 Subordinate Elements

The main model elements of TAMo described in the previous subsection are complemented by a number of other elements that allow for connections, decisions, loops and error handling. This section will introduce these elements.

Transition A transition is a directed connection between two specific model elements. For example, in a script, several task elements can be connected to create a hierarchy and thus relationship between tasks. Or in a plan, transitions are used to connect actions and states and vice versa. An element can have any number of inbound and outbound transitions. To distinguish between different outbound transitions, preconditions can be added which have to be match to follow a specific transitions. This allows for complex task hierarchies in a script or different flows of actions in a plan, thus enabling the designer to cope with different environment and agent states.

Start and End State A script as well as a plan need to have well-defined entry and exit points that depict the beginning and end of a script respectively a plan. While only a single start state can exist, several – but at least one – end states may exist.

Decision A decision element can be used to describe alternate flows of execution. If control reaches a decision element, the element's condition is evaluated and the corresponding path is selected and executed. Only one path is executed at a time but successive execution of the same decision element, e.g. in a Loop, may yield to different executed paths. Decision elements merely duplicate the feature of states to have several outbound paths that are selected based on path constraints. However, decision elements have been added to TAMo to make the graphical representation of script and plans more descriptive and easier to read.

Loop Obviously, any flow oriented model needs to support some kind of loop element that allows for the repetitive execution of certain parts of a given flow. A loop block may contain a flow of actions and states, beginning and ending with a state. Furthermore, a loop element must contains a constraint that describes when and how often the loop needs to be executed. This may be a simple boolean condition as well as a kind of loop counter that is incremented or decremented by the flow contained in the loop.

Error The introduction of special error elements into the TAMo model serves several purposes. First, it allows for concrete modeling of error cases, e.g. a failed migration, in a plan. Second, it makes it easy to distinguish between flow parts concerned with application logic and parts that handle error cases. TAMo separates between two general kinds of errors – common errors and migration errors. The former category contains all errors that may arise during the execution of an agent, e.g. during the execution of an action or a plan. The latter category encapsulates all errors that can occur during a migration from host to host. For each error, a flow of actions or a complete plan may be specified to cope with the occurred error and restore a world state that can serve as a sound base for further actions.

5.4.2 Graphical Notation

After having introduced all model elements, we will have a look at the graphical representation of those elements. We provide a graphical notation for TAMo out of two reasons: first, to allow for fast and easy sketching of agent scripts or plans using pen and paper, and second, as basis for the TAMo graphical editor that provides editing and export facilities for TAMo models. The graphical editor is covered in detail in chapter 7. For now, we will present the graphical notation for each TAMo element as well as introducing some new elements which do not alter the general TAMo model and are specific to the graphical notation.

The graphical notation is divided into two main parts: diagrams for the Task level and diagrams for the Plan level. We will start with describing all elements that occur in both types of diagrams followed by elements that are specific to either one. For each element, a figure is given that accompanies the textual specification.

5.4.2.1 Common Elements Notation

Start and End State TAMo uses the common UML representation of start and end states – a single black circle for start states and a white circle with a smaller black circle in the middle for end states.

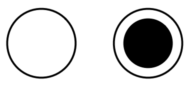


Figure 5.2: Start and End State Element

Transition A transition, that connects two other model elements, is depicted by a single directed arrow. An arrow's direction runs from the source element to the destination element.

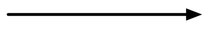


Figure 5.3: Transition Element

Constraint A constraint describes a set of conditions that must be matched in order to execute an action or follow a transition. Usually, constraints are added to transitions to allow for control during the execution of a script or to a loop block to control the execution of that loop. They are described using the well-known Object Constraint Language[2] (OCL).

Decisions There are two kinds of decisions. First, explicitly modeled decisions using the common diamond shape with a single incoming transitions and up to three outgoing transitions. Each transition is labeled with the constraint that needs to be matched in order to follow this transition. The second kind of decision is modeled more indirectly. For example, a single action may have several outgoing transitions each of them with its own constraint. If modeled this way, is is clear that the decision is made by the action itself whereas in the the former case, the decision depends on the actual outcome of an action and the given constraints at the following Decision element.

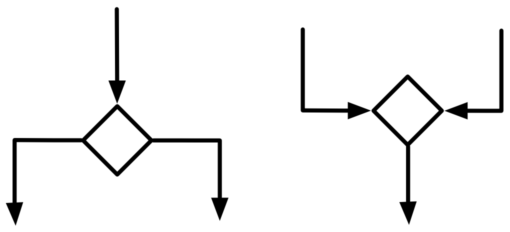


Figure 5.4: Decision Element

Loops Similar to decisions, loops can be modeled in several ways. First, one can group a set of actions inside a rectangular box to depict a recurring execution of this block. Each loop block must contain a constraint, which is shown in the upper left corner of the block, to control loop execution. A second way to model loops is to actually connect actions and transitions in a way so that the workflow's execution forms the cycle. Thus, there would be no explicit loop conditions but rather a set of states and transitions that allow for the script's circulation.

Error At various points in a script, errors may arise that can at least be named during the design process. To denote the existence of a possible erroneous outcome of an action, a yellow with a flash is used. The error case can be described using one or more constraints. The error element can be connected to two different model elements. First, it can be connected to single action to depict an error

[2]http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL

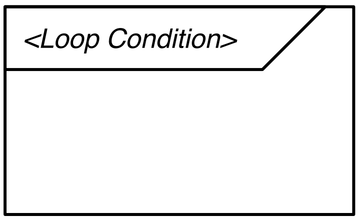


Figure 5.5: Loop Element

case that may arise during the execution of that specific action. Second, the error element can be connected to a migration element to depict an error case that occurs during agent migration.

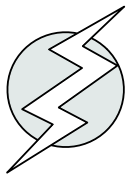


Figure 5.6: Error Element

5.4.2.2 Task Level Diagram Notation

Script There is no special representation of a script – your empty sheet of paper is simply your container for everything that makes up a script. It will contain several connected Task elements accompanied by a single start element and one or more end elements.

Task A task element is depicted by a white ellipsis with a rectangular shape containing the common name of the task at the upper left side. The ellipsis itself contains at least one square; each of those squares representing a single plan that can solve the task. Tasks may be connected with each other by transitions to depict a certain hierarchy of tasks, e.g. task A must be finished in order to attempt task B or that an agent may choose to execute task C or D because both may lead to similar outcomes.

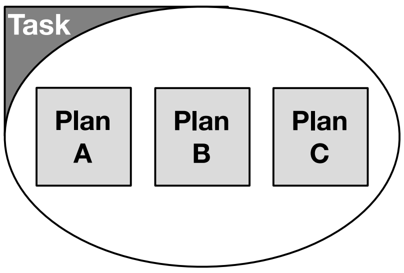


Figure 5.7: Task Element

Plan As mentioned above, a plan in a script diagram is always contained in a task element and depicted by a square with the plan's name centered inside the square. Due to the fact that a plan is rather complex and including a complete plan inside a task element is anything but feasible, we introduce a second kind of diagram to model single plans – the Plan diagram.

5.4.2.3 Plan Level Diagram Notation

State A single agent state is depicted by a rounded rectangle with a centered state name. A more elongated version of this element contains a second rounded rectangle beneath the first one which contains a formal description of the state, e.g. specific values for environment or agent parameters.

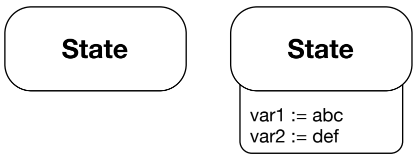


Figure 5.8: State Element

Action A single action denotes an atomic execution step and is depicted by a single rectangle with the action's name centered inside the rectangle. An action element can be connected to state elements using transition elements. Action elements may – if appropriate – also be connected to a third party module, e.g. an external database or web service to depict the usage of such an external component during an action's execution.

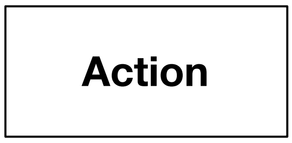


Figure 5.9: Action Element

Migration Due to the fact that agent migration is such a basic behavior of mobile agents, we decided to establish a separate graphical migration element. Visually the element is a simple dotted line that runs across at least two swim lanes and connects two states. Additionally, a migration's dotted line can be connected to an Error element to handle possible error cases.

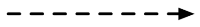


Figure 5.10: Migration Element

Swim lanes Swim lanes have been added to the graphical notation to allow for a separation of different execution hosts, e.g. agencies an agent may visit during its runtime. Similar to the UML, swim lanes are depicted as vertical rectangles with the corresponding host name centered at the top. Except for the migration element that runs across two swim lanes, each element of a plan must be contained in one single swim lane.

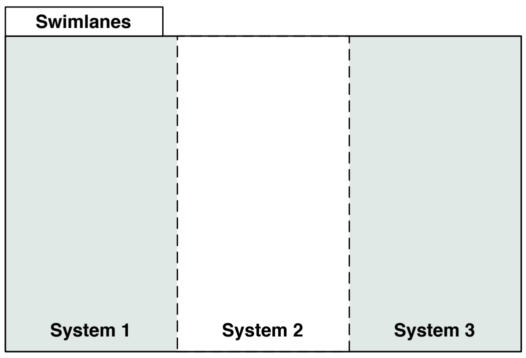


Figure 5.11: Swim Lane Element

Interfaces Some actions an agent will execute need access to third party systems like databases or web services. Therefore, we introduce a special element to model interfaces to such legacy systems and allow for their configuration by adding constraints and meta data to a connection between an action and a third party component. Such elements are depicted by the commonly know UML component element.

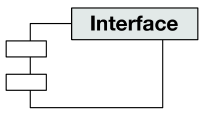


Figure 5.12: Interface Element

Comments Comments are an addition to the graphical notation of TAMo. They can be used to provide notes or hints for any TAMo model element to ease understanding of a model.

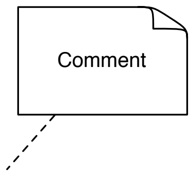


Figure 5.13: Comment Element

5.5 TAMo Revisited

After working some time with prototypes of the engine and the graphical editor, several drawbacks became apparent, which led to an adaptation and refinement of the created model. As mentioned before, TAMo Plans feature States, Transitions and Actions in whereas States and Actions are connected via Transitions in an alternating fashion to model an agent's execution flow. The separation of Actions and States was highly influenced by the Petri Net notation and seemed to be a good first step to describe an agent's plan. States should capture a distinct configuration of an agent's world state whereas Actions should be used to move from one world state to the next. However, this separation proved to be too complex and ultimately too burdensome for an easy usage of the development tools. Consider a rather simple agent that executed several actions, e.g. retrieve the URL of a remote service, access that service, evaluate the results and finally present them to the user. In the proposed model one had to create five states (a Start State, after getting the URL, after having accessed the service, after parsing the result and after delivering the final results) and four Actions with Transitions in between. So, instead of just configuring four actions, an agent designer had to create and configure 13 elements – most of them close to useless.

Thus, we decided to rethink the complete model, drop everything that seemed redundant or just remotely useless to come up with the bare minimum of elements needed to create mobile agents.

5.5.1 It's all about Actions

First of all, we removed States and Transitions; Actions are now added directly to a plan and connected with each other. Each Action is still a single, self-contained unit of work that alters the environment in a predefined way. We do, however, by no means restrict the amount of work a single Action performs. It could be a single statement of the underlying programming language like a logging message. But it could also be several hundred lines of code that access a database, filter and alter the received results and writing them back to second database. One could even move the state machine of a normal Tracy 2 agent into a single Action. An Action can define any number of outgoing connections to other Actions. Such connections represent possible outcomes and every Action has at least two of them: one for successful execution and one that should be triggered in case of an error. However, it is completely up to the programmer of an Action to provide a much more fine-grained interface to an Action, e.g. by creating one connection for each possible error case. As for incoming connections, any number of such connections may point to a single Action, e.g. we do not restrict the number of paths in a plan that end up at a specific action.

5.5.2 Implicit States

By removing State elements, we also removed the explicit notion of an agent's world state. We now have a sort of implicit States defined by an Action's input requirements and its number and kind of outgoing connections. First, an Action can only be executed if all required input values are present, thus, the moment an Action is chosen for execution marks a very specific State in an agent as the world state. Second, having an Action decide which of its many outgoing Connections is chosen seems to be a much better way of handling the flow of Actions than adding constraints to the graphical notation. Therefore, at the point where an Action decides on selecting a specific outgoing Connection, we again have a very precisely described world state.

Removing states from the model had a very welcomed effect of cutting the number of elements a user has to maintain in order to create a running plan in the editor nearly in half which should lead to a much faster creation of plans as well as visually simpler plans that are therefore easier to understand. We will come to this aspect later in chapter 7 when we cover the graphical editor.

5.5.3 Decisions, Loops, Error and Migration

The first version of TAMo featured a number of additional elements that where tailored to control the flow of a script. Namely, these were Decision, Loop and Error elements. We dropped them too. All their functionality is now provided by normal Actions. For example, TAMo comes with a Loop Action that can be configured with parameters like number of loops or a list with elements to iterate over. It also possesses, beside the Error Connection, two outgoing Connections: one that is selected while continuing with the loop and a second one that will be selected after the loop is finished. Thus, it is rather easy to integrate reoccurring sub-flows into a plan.

Compared to the old Loop Element, similar ideas have led to the removal of the Decision Element. First, every Action can provide outgoing connections for any possible outcome of this action's execution which makes it obsolete to add a cascade to decision elements to the flow after a specific Action to decide on the next steps. Second, we also provide a Decision Action that completely replaces the old Decision Element.

As for the Error element, we consider an outgoing Error Connection not as something special – it should be treated like any other Connection and we, therefore, removed the special Error element. Error Cases can be handled by the same means as anything else, e.g. one can provide a complete flow of actions that is able to compensate the negative effects of the error that occurred.

In the first installment of TAMo, a migration was handled by an interruptible Transition that contained a number of Actions for performing a migration. As mentioned above, we removed Transitions and Interruptible Transitions and were therefore in need for a compensation to provide agent migration in the new version of TAMo. We made use of the *Suspended* state for Actions to handle the local and remote part of a migration in two subsequent cycles of an agent's execution.

5.5.4 Standalone Engine

Originated as an execution engine for Tracy agents, the second version of the TAMo engine moved into a much more general direction. The base engine does not have a single reference to Tracy. Moreover, it does not have a single reference to software agents at all. One could use the engine to execute standalone scripts or programs that have been created by connecting some reusable actions. So, one could use the TAMo editor to create arbitrary programs. A version that is tailored to the specific needs of software agents as well as the Tracy agent system in particular was derived from the standalone version. It includes Actions that access Tracy features and make use of provided plugins. Furthermore, it comes with a closure agent that acts as a simple engine for TAMo model instances.

5.5.5 Other Dropped Elements

Beside the elements mentioned above, we also removed the Module and Swimlane elements because they where used for the graphical representation only and had no grounding in the TAMo engine.

5.5.6 Storage

Finally, we also made some changes to the underlying storage module. The storage is a simple associative storage that maps keys to values. In our case, keys and values are usually Strings. Every Action, Plan and Task may possess its own storage. The storage of an Action is configured by the Plan that initializes the Action. Similarly, the storage of a Plan is configured by the surrounding Task during Plan initialization. Per default, a Task respectively Plan will just hand its own storage to the newly initialized element, e.g. the Plan or Action. Thus, all elements, regardless of their abstraction level, will share the same storage which allows for easy data transfer between Actions, Plans and even Tasks. However, if security requires it, any element can hold its own, unshared storage.

5.6 Final Version of TAMo

This section will describe the elements of the final version of TAMo similar to section 5.3. Elements that haven't changed will be mentioned shortly and we will refer to their respective description in the previous section. Similarly, we will mention all elements that have been dropped and refer to their new substitutes. There may be some repetitions in the description of elements that have actually changed, but we will stress the differences to their former counterparts.

5.6.1 General Elements

Most of the general elements remained unchanged, namely the *Comment*, and *Start* and *End* elements. They still allow for marking other model elements with additional notes and, respectively, denote the beginning and end of a *Task* or *Plan*. However, the last element in the general group, *Connection*, changed significantly. Due to the fact that we dropped the *State* and *Transition* elements, the set of allowed endpoints of the Connection element has changed in the Plan diagram. In the Task Overview diagram, the definition of the Connection element did not change. It still allows for the connection of Task elements with each other and respectively Start and End elements.

5.6.2 Task Level Elements

The elements at the Task Level, namely the *Task* and *Plan* elements, remained unchanged. Still, Tasks can be combined using *Connection* elements and several Plans can be added to a specific Task element.

5.6.3 Plan Level Elements

The largest changes between the first and second version of TAMo occur at the Plan Level resulting in a much simpler model to speed up the definition of single Plans. As noted before, the *State* and *Transition* elements have been removed completely. The State of an agent is implicitly given by the current execution state of the Plan respectively Task. Furthermore, there are a no *Loop* or *Decision* elements anymore.

Instead, *Actions* are now the core building blocks of a TAMo Plan. They are self-contained elements that achieve a single purpose. Actions are connected with each other to define the flow(s) through a Plan. Any number of Connections may lead to a single Actions. However, each Action defines the number of outgoing connections and is responsible to select the appropriate outgoing connection during its execution. This differs from the usual connection handling found in workflow systems today. In todays workflow systems, the workflow designer defines the number of outgoing connections for any activity and for each connection he defines the conditon that must be matched in order to follow that connection. We have chosen to let an Action programmer define all possible outcomes because we wanted to encapsulate as much logic as possible into Actions themselves and not to expose its internals. Moreover, we overcome two problems of todays workflow approach. First, the problem of overlapping conditions where it is not exactly defined which connection should be select. Second, having an Actions outcome that cannot be matched to any given condition of all outgoing connections. Above all, we would rather not confront non-developers with the need to create boolean expressions while building an agent. They should just define, what should be done if an Action ends up in an *Error* or *No Result* state.

5.6.4 Relationship Constraints

As mentioned in several element descriptions in the previous sections, TAMo imposes a number of constraints on the relationships between different model elements. For example, it is possible to directly connect two Actions in a Plan. However, one cannot directly connect a Start to an End state which would resulting in an empty, and thus meaningless, plan.

Tables 5.1 and 5.2 depict the valid connections between any two TAMo elements as well as the possible parent-child relationships.

Element Connections	**Task Overview**	**Task**	**Plan**	**Action**	**Start- & Endpoint**	**Comment**
Task Overview	-	-	-	-	-	-
Task				-		
Plan			-	-	-	
Action						
Start- & Endpoint					-	-
Comment						-

Table 5.1: Element Connections

5.6.5 Formalization

After the introduction to the TAMo elements in a rather informal way, this section focuses on providing a sound basis for the complete model by mapping it to the notion of a nondeterministic finite automaton.

Contains Element Type	Container		
	Task Overview	Task	Plan
Task Overview	-	-	-
Task		-	-
Plan	-		-
Start- & Endpoint		-	
Action	-	-	
Comment			

Table 5.2: Element Containment

5.6.5.1 General Definitions

D	is a dictionary
D_x	is a dictionary entry
S_D	is the State of Dictionary D based on the number and and kind of its entries

5.6.5.2 Action Definitions

A	is an atomic action
C_A	is a set of actions that are connected to the outgoing connections of A, if A is the end action of a plan, then $\|C_A\| = 0$
S_A	is the state of action A with $S_A \in \{Created, Initialized, Executing, Suspended, Finished, Error, Unknown\}$
$f_A\left(S_A, S_D\right)$	a function of A such that $f_A\left(S_A, S_D\right) \rightarrow S'_A \times S'_D$, e.g. it transforms the state of action A and of the global dictionary
$g_A\left(S_A, S_D\right)$	a function of A such that $g_A\left(S_A, S_D\right) \rightarrow A'$ with $A' \in C_A$, e.g. the function selects the next action that should be executed, if A is one of the end actions, then $A' = \varepsilon$

5.6.5.3 Plan Definitions

We define a Plan P as

$$P = (A_P, S_P, A_0, E_A, D, A_c, p(A_c))$$

with

A_P	a set of actions belonging to plan P
S_P	the state of plan P with $S_P = \begin{cases} Initialized & if\, A_c = A_0 \\ Finished & if\, S_{A_C} = Finished \wedge A_C \in E_P \\ Error & if\, S_{A_C} = Error \wedge A_C \in E_P \\ S_{A_C} & if\, A_C \notin E_P \end{cases}$
A_0	the start action of plan P
E_P	a set of end actions of plan P, e.g. if the execution reaches any of these actions, the plan execution is finished after executing that very action
A_c	denotes the currently active action of plan P
$h_P(A_c)$	a function of plan P that executes f_{A_c} and afterwards g_{A_c} executed repeatedly as long as $A_c \neq \varepsilon$

5.6.5.4 Task Definition

We define a Task T as

$$T = (P_T, P_c, S_T, i_T(S_D, S_{P_c}), j_T(P_c), D)$$

with

P_T	a set of plans belonging to task T
P_c	the currently selected plan with $P_c \in P_T$
S_T	the state of task T with $T = \begin{cases} Error & if\, \forall\, P_x\, (P_x \in P_T \wedge S_{P_x} = Error) \\ S_{P_C} & otherwise \end{cases}$
$i_T(S_D, S_{P_C})$	a function of task T such that $i_T(S_D, S_{P_C}) \rightarrow P_{Next}$ with $P_{Next} \in P_T$ e.g. a function that, depending on the state of the dictionary and currently selected plan, selects the next plan for execution, P_{Next} becomes P_c
$j_T(P_c)$	a function of task T that executes h_{P_c}

5.6.5.5 Agent Script Definition

We define a TAMo Agent Script as

$$A = (T_A, T_0, E_T, T_c, S_A, k_A, l_A D)$$

such that

T_A	is a set of task that make up the scipt A
T_0	is the start task of the script A
E_T	is a set of end tasks for script A
T_c	denotes the currently selected task
S_A	the state of script A with

$$S_A = \begin{cases} Success & T_c \in E_T \wedge S_{T_c} = Success \\ Error & T_c \in E_T \wedge S_{T_c} = Failed \\ Executing & S_{T_c} = Executing \end{cases}$$

k_A	a function that executes i_{T_c} and j_{T_c} repeatedly
l_A	a function of script A such that $l_A(T_c) \rightarrow T_{next}$ with $T_{Next} \in T_A$ e.g. a function that selects the next task for execution, T_{Next} becomes T_c

5.7 Example

The final section in this chapter presents a TAMo example that is based on the scenario described in section 4.2 where an agent is in charge to determine travel options for a conference. We assume that the following information is available at the beginning of the script:

Traveller Information General information about the person who is going to visit the conference. For example hometown, preferred means of travel and time constraints derived from calendar. Furthermore, the agent should have access to several methods to contact its owner, e.g. via email, direct message or Twitter.

Conference Information The agent needs information about the conference that is going to be visited like location, date, venue notes and costs.

Budget Information Information about the overall available budget for this journey is needed to find a travel plan that fits all needs.

We further assume that the following external services exist:

Service Directory We assume that the agent has access to a general service directory where it can lookup various web services.

Hotel Booking Services We assume that the service directory provides access to web services for booking hotels. More specifically, we expect the directory to offer a worldwide booking services as well as a local service that offers accommodation options at the conference location.

Travel Option Services Similarly to the hotel booking services, we require the existence of two different services to provide options for traveling from the agent owner's hometown to the conference location, e.g. by car, train or airplane. One service could offered by a worldwide travel portal and the second one could be provided by small travel agency that is located at the owner's hometown.

Agent Hosting For all the services described above, we presume that all machines that offer these services also host an agent system that is capable to execute TAMo based agents.

Figure 5.14 shows the task layer of the TAMo script for this scenario. We have divided the agent's overall goal into four tasks and one task to handle all error cases. The first task will determine accommodation options at the conference location using one of two plans – the first one using a hotel booking service of the target city's tourist information whereas the second one uses a general worldwide booking service. The second task will determine means of travel from the agent owner's hometown to the conference and back again. Similar to the first task, the second one is backed up by two plans where each one uses a different service to determine those travel options. The third task features just a single plan which tries to match accommodation and travel options and filter them with regard to budget and possibly colliding appointments of the traveller. Task 4 is the last task in a successful execution of this TAMo script. It tries to deliver the acquired information to the agent owner either by migration to the owner's machine and present the data directly or, if migration is not an option or it simply failed, by sending it via email.

We should note that, if the third task fails, this will not lead to a failed overall script. If the agent is not able to match and filter all the available travel information, it will simply submit all the data as is to its owner. This is in contrast to the first two tasks. If one of these tasks fails, the overall script has failed. In that case, task number 5 is executed which is used to inform the owner about the current status of execution and the reasons for failure.

In figure 5.15 plan T1.1 for accessing a service, in this case a hotel booking service, is shown. The agent will first migrate to the service directory and acquire the information

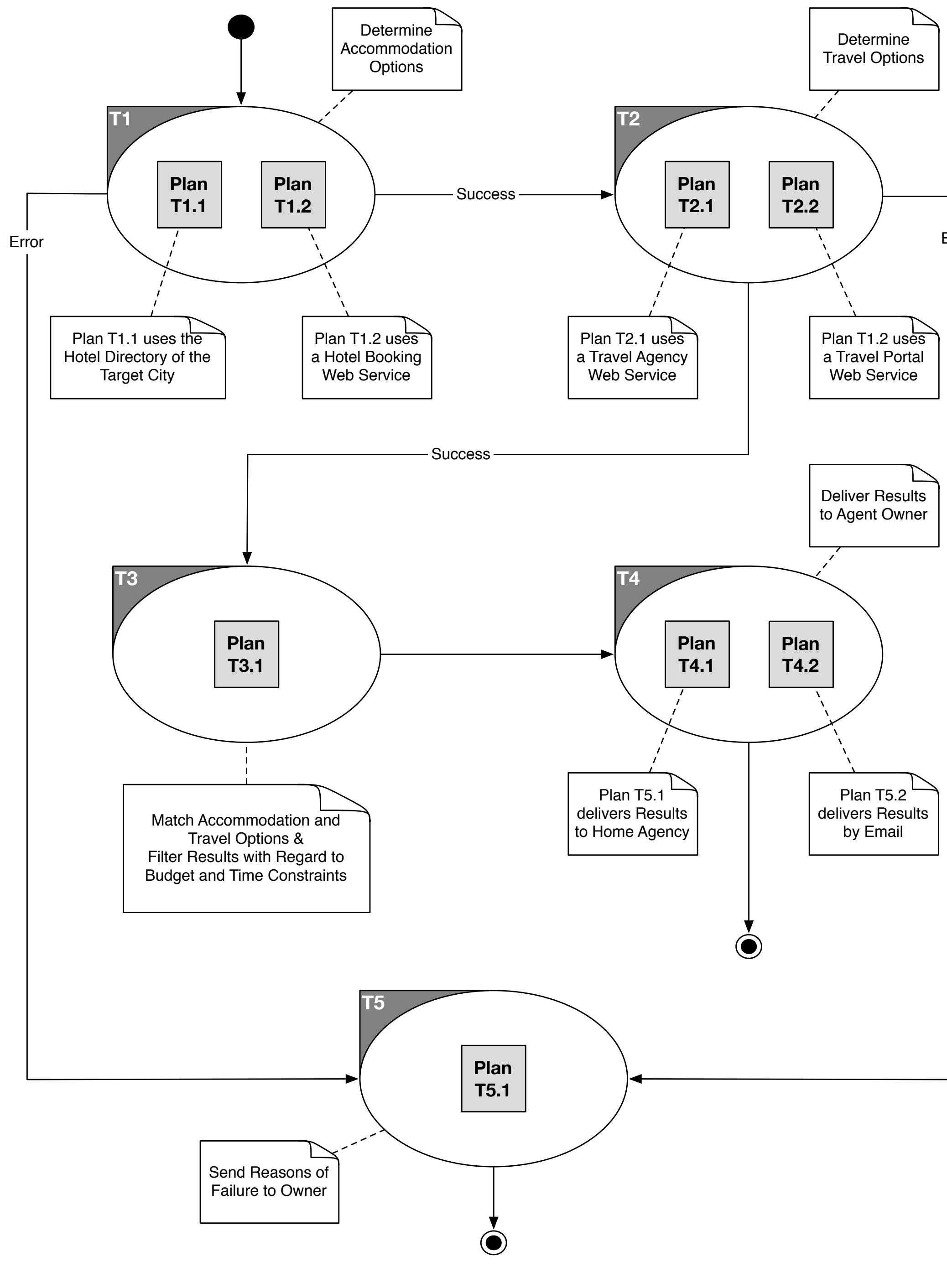

Determine Accommodation Options
T1
Plan T1.1
Plan T1.2
Success
Determine Travel Options
T2
Plan T2.1
Plan T2.2
Error
Error
Plan T1.1 uses the Hotel Directory of the Target City
Plan T1.2 uses a Hotel Booking Web Service
Plan T2.1 uses a Travel Agency Web Service
Plan T1.2 uses a Travel Portal Web Service
Success
Deliver Results to Agent Owner
T3
Plan T3.1
T4
Plan T4.1
Plan T4.2
Match Accommodation and Travel Options & Filter Results with Regard to Budget and Time Constraints
Plan T5.1 delivers Results to Home Agency
Plan T5.2 delivers Results by Email
T5
Plan T5.1
Send Reasons of Failure to Owner

Figure 5.14: Example Task Layer

about the booking service. It will afterwards migrate to that very service and use it to gather some accommodation options that will be reformatted into an internal format.

We omit the figures for plans T1.2, T2.1 and T2.2 because they are very similar to plan T1.1. They all use the service directory to find a required service which is afterwards enacted and the results are processed.

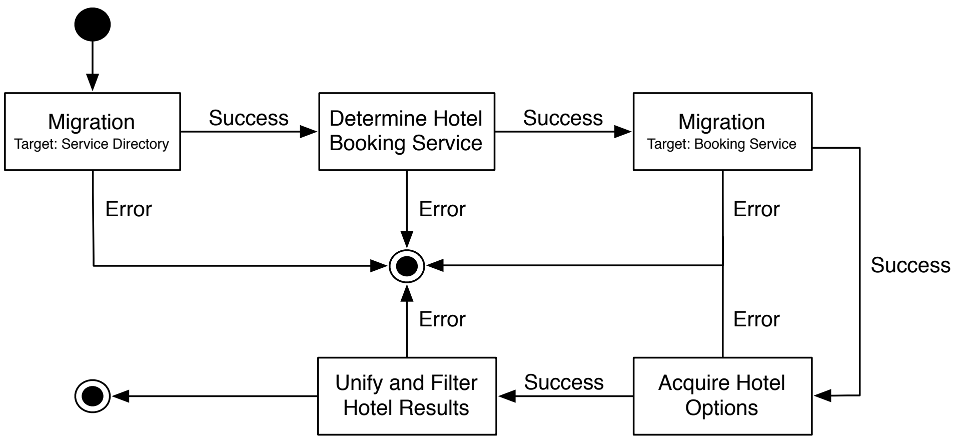


Figure 5.15: Example Plan T1.1

Plan T3.1 defines how the matching of accommodation and travel options as well as checking them against constraints is performed. For example, the action *Check Distances* will verify the distance between the hotel and the conference site complies to the owner's desires, e.g walking distance.

Both plans of task 4 are rather simple. The first one tries to migrate to the owner's home agency and present the results whereas the second one will just send an email with all the acquired information. Compare figure 5.17 and 5.18.

We also omit the figure for plan T5.1 as it is nearly equal to T4.2 - it simply sends an email to the owner detailing the reasons for failure.

5.8 Regarding the Completeness and Power of TAMo

So far, we have presented the TAMo model as well as its implementation. As we will show in Chapter 8, the created toolkit allows for easy creation and adaptation of mobile agents. Even more, with its simplicity and approachability, the framework increases the target audience for agent development and usage significantly. With these aspects

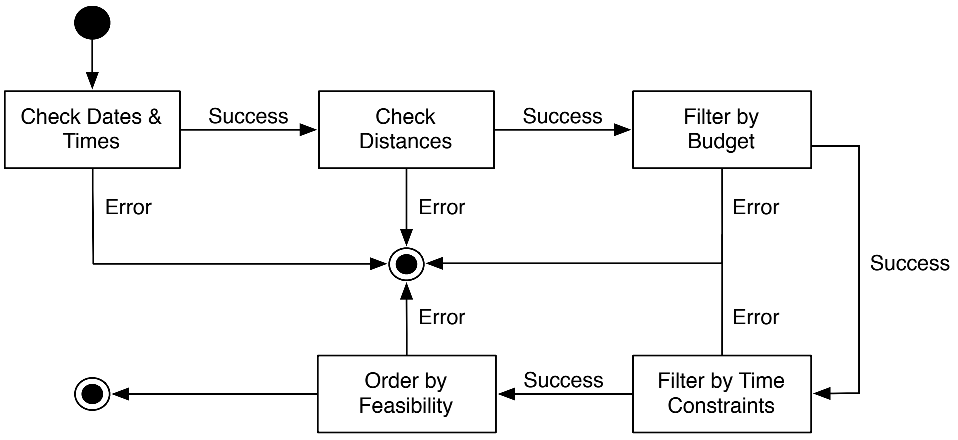


Figure 5.16: Example Plan T3.1

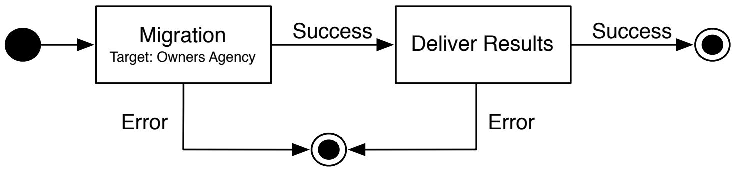


Figure 5.17: Example Plan T4.1

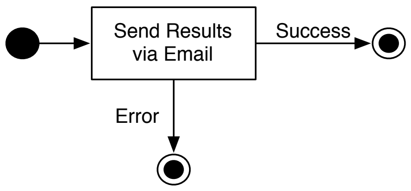


Figure 5.18: Example Plan T4.2

being the main focus of this thesis, we consider TAMo as a valuable addition to software agent research.

However, from a theoretical point of view, the presented model lacks a sound formal grounding, for example by mapping it to an existing formalisim like Petri Nets. Without such a mapping, we cannot provide more sophisticated features like validy checks, e.g. for unreachable actions or subplans, or prove the completeness of the model itself. But these aspects have not been focus of this work and we leave it to future research. And, concerning the computation power and completeness of TAMo, we regard the fact that the implemented framework is grounded on the Java programming language and allows for the usage of all language features as sufficent enough for our aims.

Chapter 6

Implementation of the TAMo Runtime Engine

This chapter covers the implementation of the TAMo engine. We will start with a general overview of the goals and constraints that guided the development of the TAMo execution environment and how the model described in the last chapter was changed to allow for a better implementation. Thereafter, we take deeper looks at the various parts that make up the final system. Afterwards, in chapter 7, the graphical editor which accompanies the core system and allows for an easy creation of tasks and plans will be discussed.

6.1 Overview

With the final model described in the last chapter, we started to conceive a flexible and easy to understand implementation. In a simple 1:1 fashion, we first translated every model element into a concrete element in our implementation and afterwards tried to fill in the gaps and holes as well as to remove one or the other element out of convenience or sheer optimization.

We will first outline the implementation of the general TAMo engine that has no references to the Tracy 2 agent toolkit and that is able to run standalone execution scripts. Afterwards, the integration of TAMo as the new agent model and execution engine for Tracy 2 is described.

6.2 The Standalone Engine

The core TAMo engine, that is based on the TAMo model, was implemented as a standalone runtime for the execution of TAMo based scripts. One of the main goals of the implementation was to restrict the number of dependencies to other frameworks or libraries to a bare minimum. Fortunately, we were able to ground the core engine solely on the standard libraries provided by the Java programming languages that was used for the implementation and have no other dependencies to third party frameworks or APIs. Thus, it is possible to easily integrate the core TAMo engine into any software that is based on the Java runtime environment. And, with the integration into the Tracy 2 agent toolkit, we were the first to benefit from these minimal requirements.

The implementation is based around the four core parts of the TAMo model: *Tasks*, *Plans*, *Actions* and the shared *Data Storage*. Therefore, for each of these elements an interface as well as a default implementation respectively an abstract base class are provided. In the following subsections, we will have a closer look at these parts.

6.2.1 Shared Elements

In this section, we will describe those parts of the TAMo engine that are shared among the other elements.

ExecutableState We have provided an Java *Enum* that captured the different states that can be adopted by Tasks, Plans, Actions and a TAMo script as a whole. The possible states are *CREATED*, *INITIALIZED*, *EXECUTING*, *SUSPENDED*, *FINISHED*, *ERROR* and *UNKNOWN*. Figure 6.1 shows the possible state transitions.

CoreEngine This class represents the outer wrapping around any TAMo script and serves as the starting point for any execution. The *CoreEngine* is given a set of interconnected Tasks and will execute them in the defined order by successively calling the corresponding *run* method of a Task.

TAMoLogger To capture how the execution of a TAMo script performs, we created a simple logger that writes status updates to a console. We are aware that there are a lot of sophisticated Logging frameworks available, but due to the fact that we wanted to have a minimal set of dependencies, we decided to create a small one especially for TAMo. And, with a logger having such a small footprint, the burden

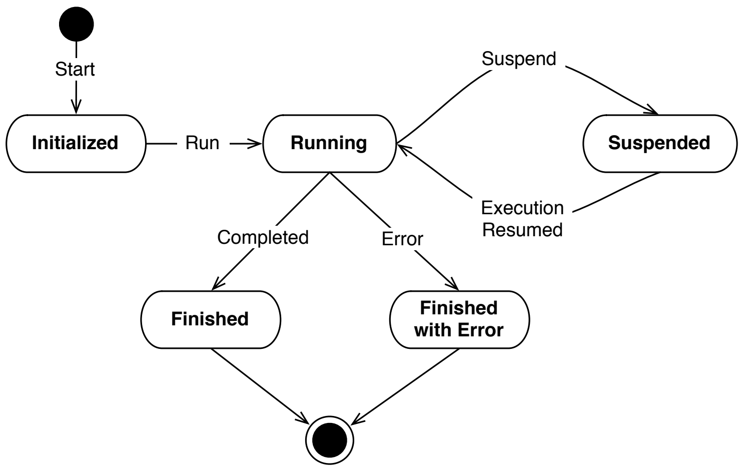


Figure 6.1: States Transitions as implemented for Tasks, Plans and Actions

of carrying this logger during a migration is negligible. Moreover, extending the TAMoLogger to wrap a more sophisticated Logging framework is straightforward.

6.2.2 Task Implementation

6.2.2.1 Interfaces

ITask The *ITask* interface defines the structure and external functionality of a TAMo Task. Beside five methods to handle state transitions, for example *execute* or *suspend*, it offers methods to add and remove plans and to set a storage, plan selector and delegate implementation.

ITaskDelegate During the execution of a task, several state transitions take place and to allow for tracking those changes, we have implemented the Delegate pattern [Buck and Yacktman, 2009]. The *ITaskDelegate* interface defines a set of methods that a task delegate must support in order to follow the execution of a task. In general, for each state transition, the delegate is informed two times. First, before the task will initiate this transition and second, after the transition took place. Moreover, the delegate will be informed when the currently executed plan has failed and when a new plan will be and was selected.

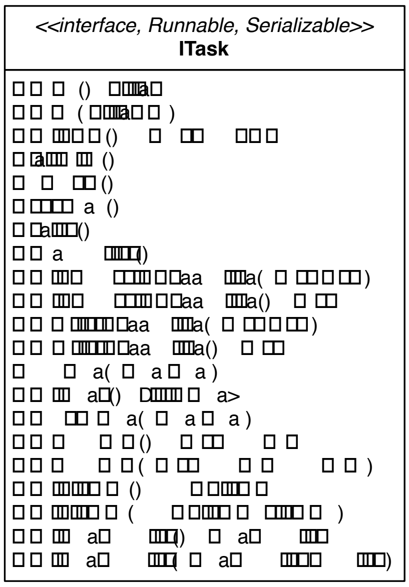


Figure 6.2: ITask Class Diagram

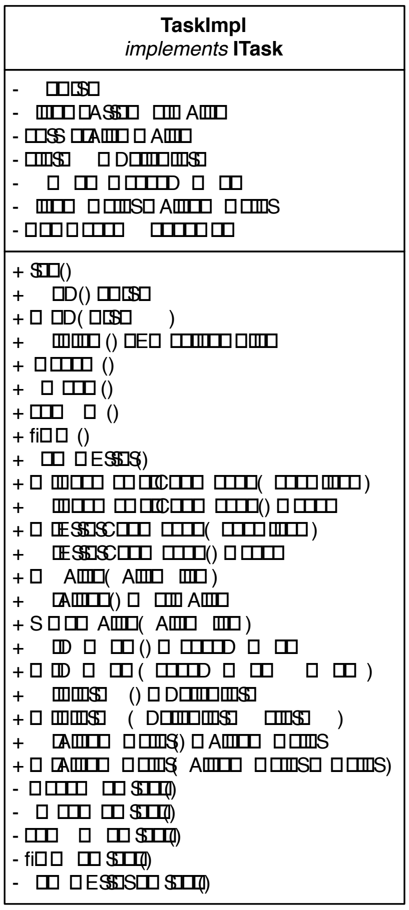
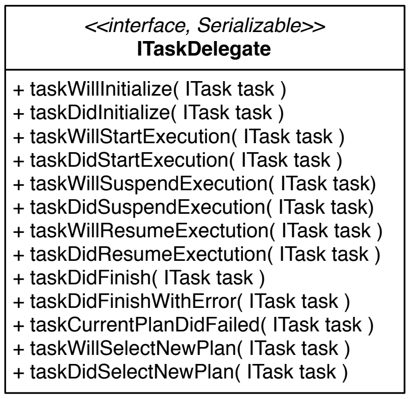


Figure 6.3: TaskImpl Class Diagram

IPlanSelector In order to provide different means to handle the selection of one of the provided plans that are associated with a single task, we decided to apply the *Strategy* pattern [Gamma et al., 1994] and move the selection process into separate classes. The *IPlanSelector* interface defines the required functionality that a plan selector must provide. To keep things simple, there are just two methods. A first method that determines if there are any executable plans left, e.g. at least one plan with a state other than *Failed*. And a second method that returns the next plan that should be executed.

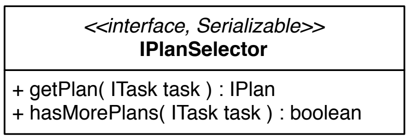
6.2.2.2 Classes

TaskImpl This class represents a straightforward implementation of the *ITask* interface. It handles a task's states and transitions, plan selection, configuration and execution as well as informing an optional delegate about its execution. This implementation should be sufficient enough for nearly any usage of the TAMo engine and altering the behavior of this class should only be necessary in extreme border cases. Using a different plan selector or providing a delegate that hooks into the execution should provide enough options to alter the behavior of this implementation.



<<interface, Serializable>>
ITaskDelegate
+ taskWillInitialize(ITask task)
+ taskDidInitialize(ITask task)
+ taskWillStartExecution(ITask task)
+ taskDidStartExecution(ITask task)
+ taskWillSuspendExecution(ITask task)
+ taskDidSuspendExecution(ITask task)
+ taskWillResumeExectution(ITask task)
+ taskDidResumeExectution(ITask task)
+ taskDidFinish(ITask task)
+ taskDidFinishWithError(ITask task)
+ taskCurrentPlanDidFailed(ITask task)
+ taskWillSelectNewPlan(ITask task)
+ taskDidSelectNewPlan(ITask task)

Figure 6.4: ITaskDelegate Class Diagram



<<interface, Serializable>>
IPlanSelector
+ getPlan(ITask task) : IPlan
+ hasMorePlans(ITask task) : boolean

Figure 6.5: IPlanSelector Class Diagram

SimplePlanSelector For our testing purposes, we created a very simple plan selector. Without considering any environment parameters, this selector will simply select the next unfinished plan for execution.

LoggerTaskDelegate To track the execution of TAMo scripts and measure runtimes, we created a simple task delegate that captures timestamps for all state transitions and uses the TAMoLogger to print these values.

6.2.3 Plan Implementation

6.2.3.1 Interfaces

IPlan The *IPlan* interface defines the structure of a TAMo plan implementation. Similar to the *ITask* interface, it offers methods to handle state transitions and a storage. Furthermore, it provides means to set and get the start action of the plan as well as get the currently executed action. To monitor plan execution, we also applied the *Delegate* pattern. Analog to the task delegate, the plan delegate is informed of any upcoming and executed state transition.

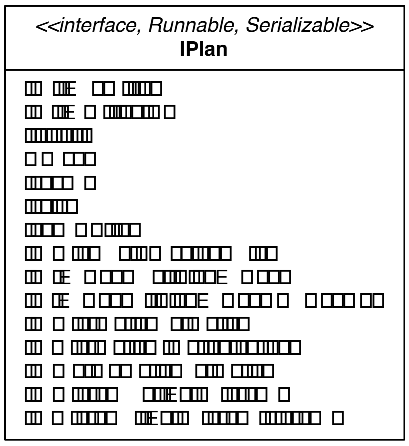


Figure 6.6: IPlan Class Diagram

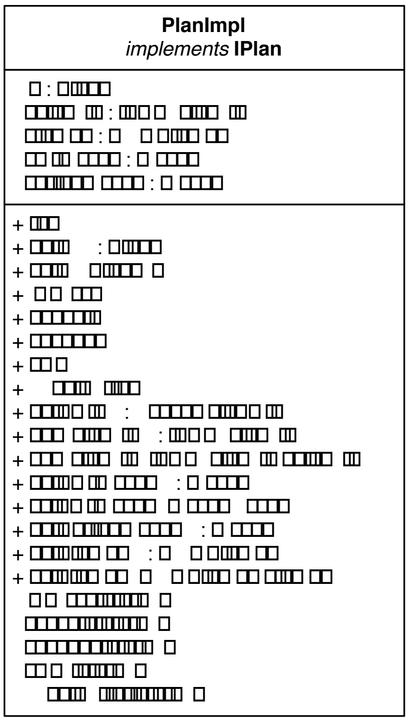


Figure 6.7: PlanImpl Class Diagram

IPlanDelegate Similar to the *ITaskDelegate* interface, the *IPlanDelegate* interface defines the methods that a class must implement to act as a plan delegate and monitor a plan's execution.

6.2.3.2 Classes

PlanImpl The *PlanImpl* class provides a straightforward implementation of the *IPlan* interface and should be sufficient enough for most application scenarios. Similar to our standard task implementation, it handles a plan's states and the corresponding transitions, action configuration and execution as well as keeping an optional delegate informed on its current execution status.

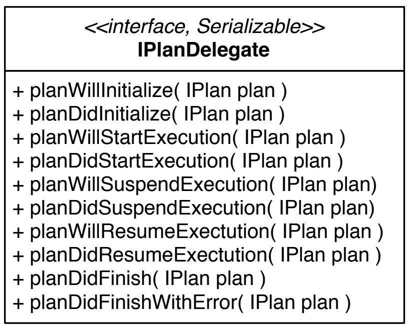

<<interface, Serializable>>
IPlanDelegate
+ planWillInitialize(IPlan plan)
+ planDidInitialize(IPlan plan)
+ planWillStartExecution(IPlan plan)
+ planDidStartExecution(IPlan plan)
+ planWillSuspendExecution(IPlan plan)
+ planDidSuspendExecution(IPlan plan)
+ planWillResumeExectution(IPlan plan)
+ planDidResumeExectution(IPlan plan)
+ planDidFinish(IPlan plan)
+ planDidFinishWithError(IPlan plan)

Figure 6.8: IPlanDelegate Class Diagram

LoggerPlanDelegate To track the execution of plans and perform runtime measurements, we created a simple plan delegate which captures timestamps for all of a plan's state transitions and submits them to the global *TAMoLogger*.

6.2.4 Action Implementation

6.2.4.1 Interfaces

IAction The *IAction* interface defines the structure of a valid TAMo action implementation. Similar to the *ITask* and *IPlan* interfaces, it features the same methods to handle state transitions as well as defining a delegate; in this case an object of type *IActionDelegate*. Due to the fact that a single TAMo action can offer any number of outgoing connections, the interface defines methods to link follow up actions to these connections and to retrieve such linked actions. A rather large set of methods of the *IAction* interface is concerned with the manipulation of the data storage. There are two different options to handle data storage access: by value or by reference. The first one is used for entries that are specific to that action, for example action configuration. In this case, the action uses a dictionary key to access the concrete value of a parameter. The latter type, by reference, can be used to exchange data between different actions. Here, the action uses a key to access the data storage and acquire a second key. With the second key, the action can access the actual value of the parameter. Thus, it is possible to have several actions access the same data in the data storage without linking them during the implementation. The definition of shared keys is thus postponed until the actual creation of a TAMo script.

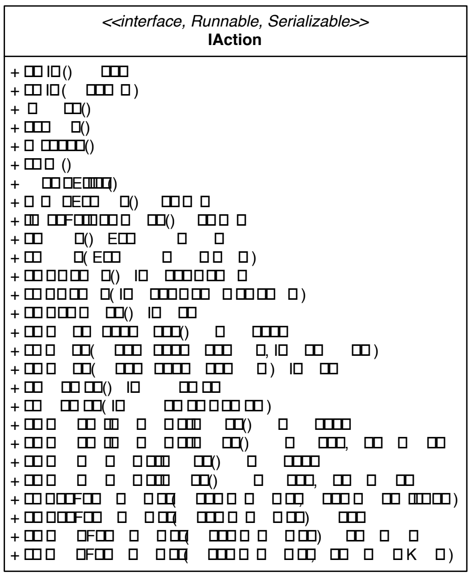


Figure 6.9: IAction Class Diagram

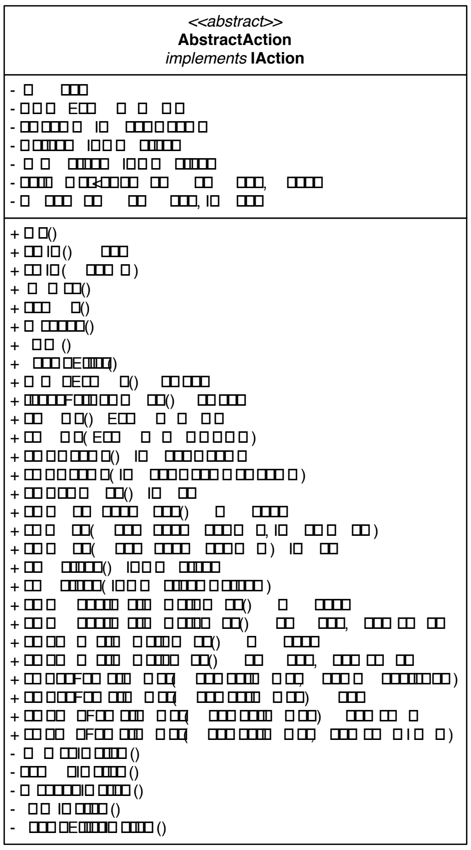
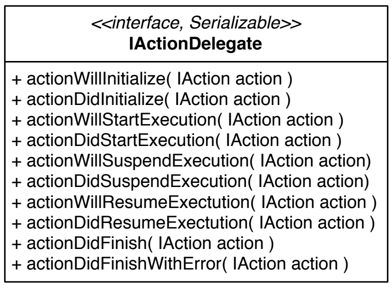


Figure 6.10: AbstractAction Class Diagram

IActionDelegate The *IActionDelegate* provides the same functionality as the delegates for Tasks and Plans. It allows for monitoring the state transitions that take place during an actions execution.

6.2.4.2 Classes

AbstractAction This class is an abstract implementation of the *IAction* interface that provides all the basic functionality any TAMo action should offer, e.g. state handling, data storage access and notifying the delegate. It should serve as the base class for any concrete TAMo action. By inheriting from this class, all an action developer needs to do is provide an implementation of the *execute()* method and define necessary parameters and outgoing connections. By default,



```
<<interface, Serializable>>
IActionDelegate

+ actionWillInitialize( IAction action )
+ actionDidInitialize( IAction action )
+ actionWillStartExecution( IAction action )
+ actionDidStartExecution( IAction action )
+ actionWillSuspendExecution( IAction action)
+ actionDidSuspendExecution( IAction action)
+ actionWillResumeExectution( IAction action )
+ actionDidResumeExectution( IAction action )
+ actionDidFinish( IAction action )
+ actionDidFinishWithError( IAction action )
```

Figure 6.11: IActionDelegate Class Diagram

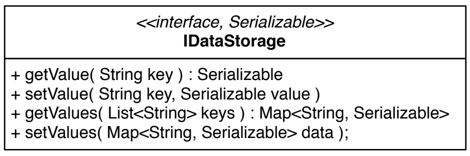
AbstractAction already offers two outgoing connections for the success and error cases.

LoggerActionDelegate Similar to the logger delegates for tasks and plans, we created a delegate for actions that would capture state transitions and provide runtime measurements.

6.2.5 Data Storage Implementation

6.2.5.1 Interfaces

IDataStorage The data storage that is associated with any TAMo script is a straightforward key-value based data structure. Therefore, the *IDataStorage* interface defines the common methods to access such a kind of storage.



```
<<interface, Serializable>>
IDataStorage

+ getValue( String key ) : Serializable
+ setValue( String key, Serializable value )
+ getValues( List<String> keys ) : Map<String, Serializable>
+ setValues( Map<String, Serializable> data );
```

Figure 6.12: IDataStorage Class Diagram

6.2.5.2 Classes

DataStorageImpl The *DataStorageImpl* class provides a default implementation of the *IDataStorage* interface which is backed up by a standard Java *Map* data structure.

6.2.6 Component Interaction

Diagram 6.13 displays the interactions that take place between the parts which make up the core TAMo engine and it should serve as a reference for anyone who is going to use and alter the framework.

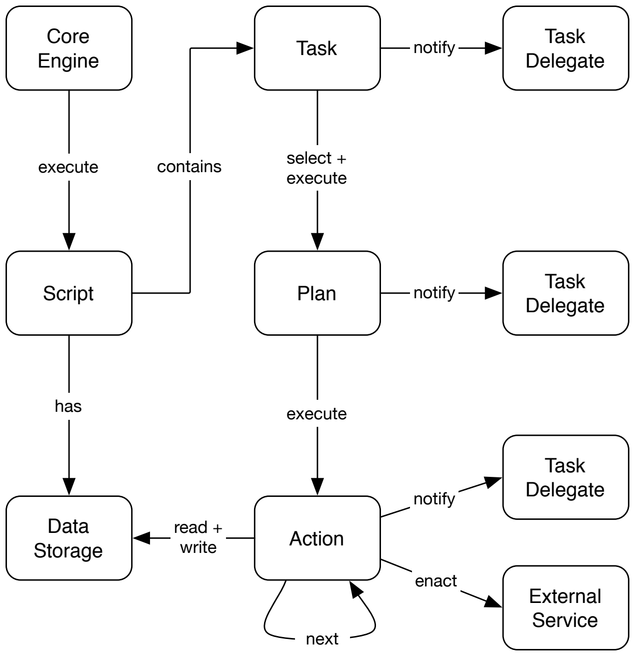


Figure 6.13: TAMo Component Interaction

6.3 Integration of TAMo into Tracy 2

In this section, we will take a look at the integration of the TAMo engine into the Tracy 2 agent toolkit to serve as the new agent execution environment. As stated above, due to its minimal dependencies, this process was simple. Basically, we transformed the *CoreEngine* into a normal Tracy 2 agent. Thus, we got a generic agent that is able to execute any TAMo script. Beside that, we create some additional classes to handle access to specific Tracy functions and extended the interfaces and classes described in the last chapter with regard to agent related properties. The general structure of a Tracy 2 agent system with an integrated TAMo engine can be seen in figure 6.14

whereas in figure 6.15, usage and dependency relations between the core parts of Tracy and the new TAMo elements are displayed.

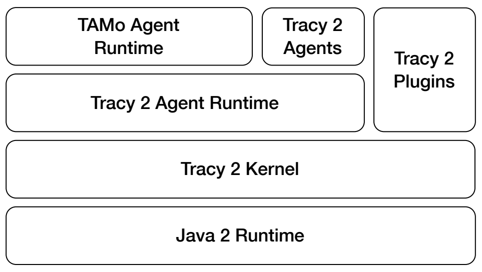


Figure 6.14: TAMo and Tracy 2

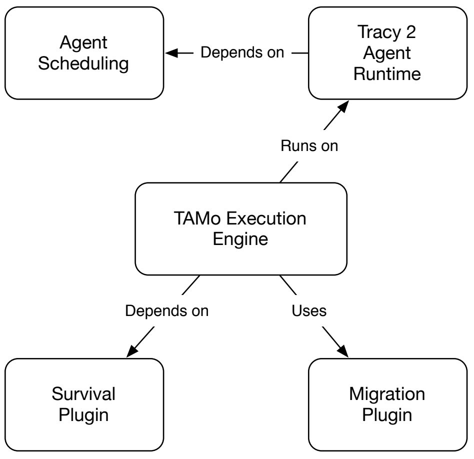


Figure 6.15: Usage and Dependencies

In the following subsections, we will have a closer look at the updated and new parts of TAMo which have been developed during the integration into the Tracy 2 agent system. Figure 6.16 presents the complete class diagram of the TAMo version for the Tracy 2 agent toolkit and can serve as reference for the upcoming sections.

6.3.1 Agent and Agency

In this section, we describe interfaces and classes that have been created to integrate the TAMo framework as an execution engine into the Tracy 2 agent toolkit alongside the traditional agent programming environment.

6.3.1.1 Interfaces

IAgent The *IAgent* interface has been created to enable a *Task*, *Plan* or *Action* to access agent and agency specific features. So far, it offers access to the *Plugin-ContextHandler*, migration functionality as well as requesting a relaunch of the agent. In section 6.3.2 and 6.3.3, we outline the changes that have been made to the *ITask*, *IPlan* and *IAction* interfaces as well as their respective implementations to integrate a reference to an *IAgent* implementation.

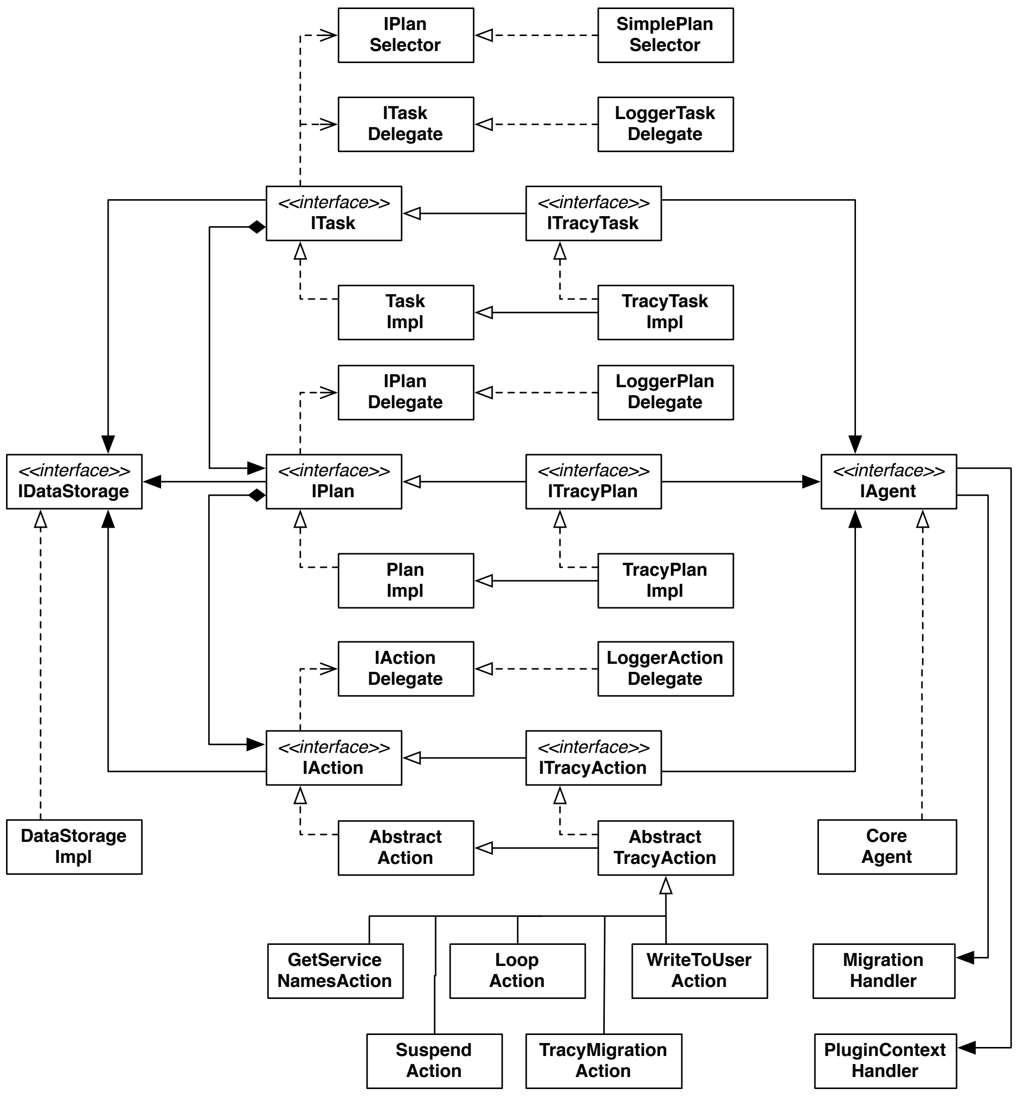
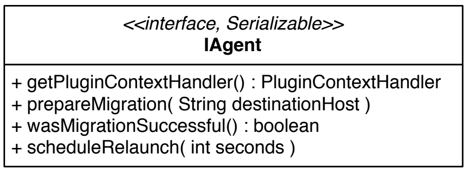


Figure 6.16: Tracy and TAMo Class Diagram

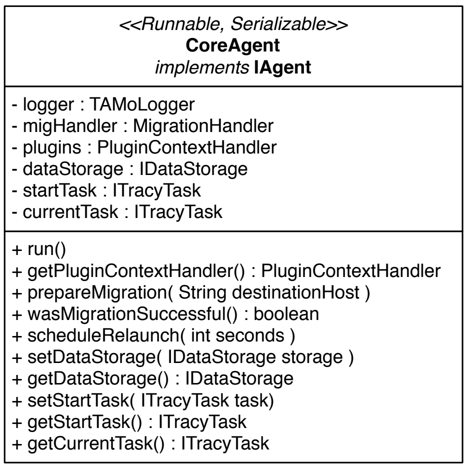
6.3.1.2 Classes

CoreAgent The *CoreAgent* is a standard Tracy 2 agent which serves as a wrapper around the core TAMo engine. Thus, it is able to execute any TAMo script inside of Tracy 2. By choosing this approach, we could introduce TAMo into the Tracy 2 agent system without altering the standard agent programming approach or breaking any existing agent respectively agent application. The *CoreAgent* class implements the *IAgent* interface and will take care of loading a given TAMo script, initialize its components, provide access to agency specific plugins as well as offering a simple access to agent migration. Furthermore, it will control the life cycle of a TAMo script and reschedule the agent for another execution of there are still tasks to do by using the *Survival* plugin offered by Tracy.



<<interface, Serializable>>
IAgent
+ getPluginContextHandler() : PluginContextHandler
+ prepareMigration(String destinationHost)
+ wasMigrationSuccessful() : boolean
+ scheduleRelaunch(int seconds)

Figure 6.17: IAgent Class Diagram



<<Runnable, Serializable>>
CoreAgent
implements IAgent
- logger : TAMoLogger
- migHandler : MigrationHandler
- plugins : PluginContextHandler
- dataStorage : IDataStorage
- startTask : ITracyTask
- currentTask : ITracyTask
+ run()
+ getPluginContextHandler() : PluginContextHandler
+ prepareMigration(String destinationHost)
+ wasMigrationSuccessful() : boolean
+ scheduleRelaunch(int seconds)
+ setDataStorage(IDataStorage storage)
+ getDataStorage() : IDataStorage
+ setStartTask(ITracyTask task)
+ getStartTask() : ITracyTask
+ getCurrentTask() : ITracyTask

Figure 6.18: CoreAgent Class Diagram

MigrationHandler The *MigrationHandler* was introduced into the TAMo framework to allow for an easy usage of agent migration and take care of releasing plugin contexts and, because it would otherwise prevent a migration, signing out of the *Survival* plugin. By offering such a simple access to the migration engine, we are hiding many of the more sophisticated features. However, our approach offers an easy and fast start into applying migration in TAMo scripts and, by implementing additional Actions that can be used in a TAMo script, the usage of advanced migration options like strategies or code and mirror servers is still possible.

PluginContextHandler As described in section 2.6, most high-level functionality in Tracy 2 is provided by plugins that offer plugin-specific context object to access their services. Those context objects can be acquired by using a static method of the *Context* class, which works well for agents that consist of only a single class and can hold references to contexts they need to access several times. However, in TAMo, functionality and thus access to plugins is distributed among a number of actions. To prevent the TAMo model from requesting plugin contexts over and over again during execution, we established a *PluginContextHandler* class that acts as mediator between a TAMo instance and a Tracy context object. Beside decreasing the coupling between Tracy 2 and TAMo, the *PluginContextHandler* will also cache requested context objects for faster succeeding access.

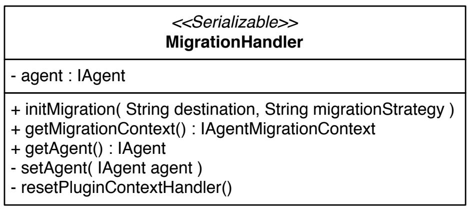

<<Serializable>>
MigrationHandler
- agent : IAgent
+ initMigration(String destination, String migrationStrategy)
+ getMigrationContext() : IAgentMigrationContext
+ getAgent() : IAgent
- setAgent(IAgent agent)
- resetPluginContextHandler()

Figure 6.19: MigrationHandler Class Diagram

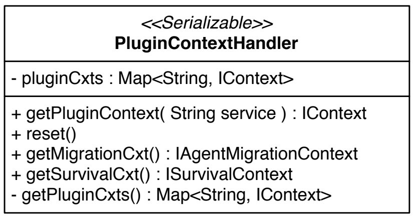

<<Serializable>>
PluginContextHandler
- pluginCxts : Map<String, IContext>
+ getPluginContext(String service) : IContext
+ reset()
+ getMigrationCxt() : IAgentMigrationContext
+ getSurvivalCxt() : ISurvivalContext
- getPluginCxts() : Map<String, IContext>

Figure 6.20: PluginContextHandler Class Diagram

6.3.2 Tracy Actions

Based on the basic interface and implementation for a TAMo action, specialized versions for the Tracy 2 integration have been created to ease the usage of agent and agency specific features. After outlining these changes, we describe a set of different actions that have been created for the utilization in a Tracy 2 TAMo script.

6.3.2.1 Interfaces

ITracyAction Extending the *IAction* interface, the *ITracyAction* adds a single method to submit an *IAgent* reference to an action. Thus, such an action is able to access the features provided by an agent as describes in section 6.3.1.2.

6.3.2.2 Classes

AbstractTracyAction The *AbstractTracyAction* extends from *AbstractAction* and implements the *ITracyAction* interface thereby adding a reference to an *IAgent* implementation. Similar to its parent class, this class should serve as the base class for all Tracy 2 TAMo actions.

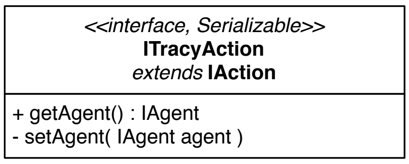


Figure 6.21: ITracyAction Class Diagram

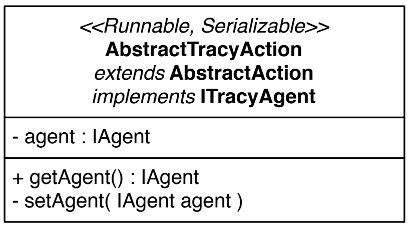


Figure 6.22: AbstractTracyAction Class Diagram

6.3.2.3 Available Actions

GetServiceNamesAction This action can be used by an agent to acquire a list of services that are currently available at the platform. Given such a list, an agent can select an appropriate service for the task at hand which can afterwards be enacted.

SetPersistencyCheckpointAction As part of the MobiSoft project, the Tracy team developed a plugin for Tracy 2 agencies that enabled agents to save their current state, shut down and resume execution at some point in the future. Saving and restoring is handled by the plugin; an agent just needs to register at this plugin. The Persistence plugin was mainly targeted to allow for a gentle agency shutdown and restart without losing running agents during this process. This TAMo action allows for agents to easily use this service.

RESTServiceAccessAction This is a generic action that enables agents to access a RESTful web service (compare section 2.3.2) by providing easy to use abstractions for setting an HTTP method, header parameter or the body of such a request. The reply is handed directly to the agent for any further usage.

WriteToUserAction A very simple action used mainly for debugging purposes. It allows for sending a messages to the user that is currently logged in at the Tracy 2 shell.

LoopAction In our move from the first to the second, final version of TAMo, we removed several special elements like the loop construct. To compensate for this loss, we created an action that serves the same purpose. It is possible to configure a *LoopAction* with a fixed number of iterations or supply a list based data structure for iteration. Beside the default outgoing success and error connections, it offers a loop connection which is chosen if the iteration should continue.

SuspendAction Due to their asynchronous nature, agents will often wait for some external event like an incoming message or a reconnection of their current agency to a network. Therefore, we created an action that will pause an agent for a specific amount of time.

TracyMigrationAction The*TracyMigrationAction* triggers a simple Push migration by using the functionality offered by the *CoreAgent*. Everything that belongs to the agent, e.g. code, data, and state, will be send to the destination agency. The destination can be submitted with the actions configuration or it is acquired from the data storage during runtime.

6.3.3 Tracy Tasks and Plans

Similar to the *IAction* and *ITracyAction* interfaces, the *ITask* and *IPlan* interfaces have been extended to allow for supplying an *IAgent* reference to a task or plan. Further, the *TaskImpl* and *PlanImpl* have been extended in a similar manner to integrate the new interface. Beside that, no changes to these implementations have been made.

6.3.4 Agent Lifecycle

This section will describe the general lifecycle of a TAMo based agent. As depicted in figure 6.23, the agent's execution starts with a standard initialization procedures and the first task of the given script is selected for execution. The plan selector is issued to select the most appropriate plan for the actual task. Afterwards the execution of the chosen plan is started and the first action contained in the plan is executed. After this execution, the *CoreAgent* decides on the next step – if there are still actions to perform or tasks to complete, the agent will schedule a reenactment of the agent at the *Survival* plugin. At this point, the first invocation of the agent's *run* method ends. Depending on the parameters given to the *Survival* plugin, the agent will rerun instantly or after some given delay. This second invocation will not be bothered with any initialization task as these have already been conducted during the first cycle. The second invocation will simply continue the execution of the currently active task or plan respectively select a new task or plan if the previous ones where finished. The execution of the next action concludes the second invocation of the agent's *run* method. Again, the *CoreAgent* will decide on a rescheduling of the agent.

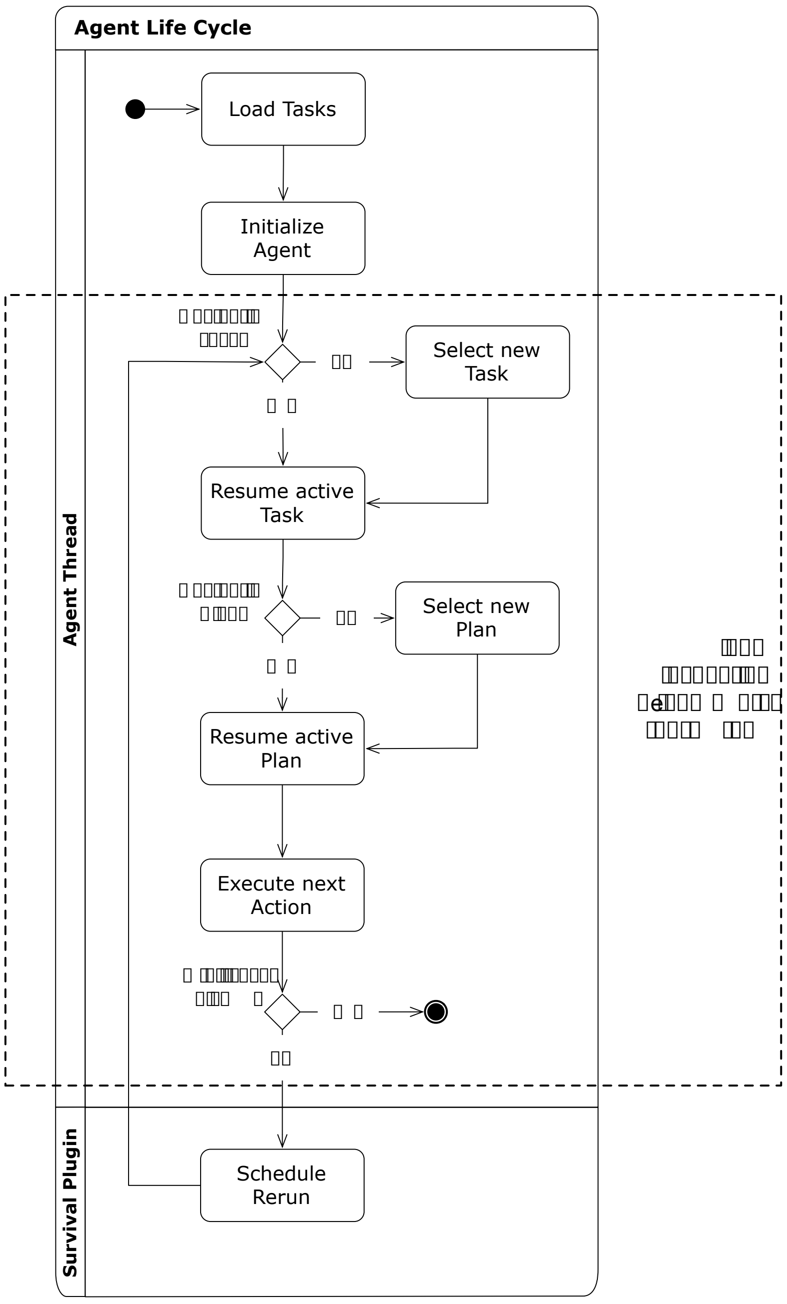


Figure 6.23: TAMo Agent Life Cycle

Chapter 7

Implementation of a Graphical Model Editor for TAMo

Beside creating a new, easy to use development and execution framework for mobile agents in terms of a formal notation and a reference implementation, we also aimed at providing easy to use tools that allow for the creation of agent execution scripts for programmers, designers and, ideally, end users. With TAMo incorporating concepts of workflow systems, the kind of tool to help in the development of TAMo models should be similar to graphical editors used in workflow system to create workflows by simply adding activities to a sketch-board and connecting those activities in the proper execution order. Editors like these have been used for years and are easy to understand and utilize, making the core concept a good starting point for graphical TAMo model editor. Furthermore, it should be possible to use the editor as a standalone program – the preferred way for an agent designer or end user – but also in conjunction with a development tools, where an agent programmer can create new activities and test them without switching her well-known environment. The standalone version could also be used in discussions with customers to explain and adapt a model to their needs as a graphical notation can greatly help non-experts to understand the innards of the software. To enable the creation and usage of a standalone version of the editor, the coupling between the TAMo model framework and the editor should be as weak as possible with only a handful of dependencies.

The editor itself must provide the following functionality:

- Create, edit, save and delete Task diagrams
- Create, edit, save and delete Plan diagrams

- Dynamically load Actions
- Configure Actions
- Configure/Pre-fill the Data Storage
- Export final models in an interchangeable format that can be read by the TAMo engine

7.1 Foundation

We decided to implement the TAMo Editor as a plugin for the Eclipse Platform[1]. Eclipse is a widely used, Open Source Integrated Development Environment (IDE) for a large number of languages and it is the predominant IDE for Java development. But Eclipse is more than just a development environment. Based on an OSGi[2] architecture, Eclipse features a core runtime that can be adapted to any imaginable requirement by the addition respectively extension of plugins. The OSGi specification defines an architecture that is capable of Hot Plugging, a technique to dynamically load and update software modules at runtime. The OSGi specification calls such software modules *bundles* and every Eclipse plugin conforms to the OSGi bundle specification. For more information on the OSGi specification we refer to the literature [McAffer et al., 2010].

The core Eclipse system as well as its plugins provide *Extension Points* which can be used to hook plugins into the system. A plugin can extend the core functionality of Eclipse in various ways, for example by integrating new programming languages, adding new kinds of editors like the one we have in mind or adding access to remote systems and external tools. Interestingly, all functions that make up the Eclipse IDE are provided by plugins which run on the core OSGi platform.

Due to its wide spread among developers, using the Eclipse platform for our editor is the natural choice for the agent programmer part. Beside that, the Eclipse platform allows for the creation of standalone applications that consist of the core OSGi platform and a selected set of plugins which make up an application. So, by using the Eclipse platform, creating a standalone Editor application for agent designers and end users can be accomplished as well. Furthermore, in the long term, we aim at integrating the TAMo editor into the Tracy 2 Administration *Wai Lin* (see section 2.6.3).

[1]http://www.eclipse.org
[2]http://www.osgi.org

7.1.1 Editors and Views

An Eclipse plugin has two general types of elements – *Editors* and *Views*. Editors allow for the manipulation of the underlying data whereas Views provide additional information. Compared to the Eclipse-based Java Development Environment, Editors allow for the direct manipulation of program code and provide features like code coloring or code completion. Furthermore, they provide a sound handling for a *saved* and *unsaved* status of the manipulated model element. Views however display additional data for the currently edited file, like to outline of the structure of a class, or the complete project, like the list of all files that belong to it. But the separation between these two elements is not as strict as it seems – the so called *Properties View* allows for the manipulation of (meta) data of the currently edited model. Depending on the underlying model, an editor can take every shape – from a simple text editor for code manipulation to a form-based representation of complex configuration files up to a sophisticated graphical representation of an underlying model.

7.1.2 Graphical Editing Framework

The TAMo Editor uses several third-party plugins that are available and which provide a sound starting point for creating a graphical editor. The first of those plugins is the *Graphical Editing Framework* (GEF)[3] plugins which supports the creation of graphical editors following a strictly Model-View-Controller Paradigm (MVC):

Model A GEF model contains and maintains all data that is manipulated by the user and has no references to any other part of the editor. Changes to the model will be propagated to one or more controllers, which then update the views accordingly. It is highly advisable to create a sound model implementation before starting with any other part of the editor. In case of TAMo, the model implementation of the engine was used as basis and extended when necessary. GEF imposes several constraints on a valid model; most of them are targeted to establish a solid notification mechanism for all parts of the editor which have been incorporated into the TAMo engine model. During runtime, the creation of model elements is accomplished via Factory objects [Gamma et al., 1994] which link the new element to all its dependencies.

Controller A GEF controller links exactly one model element with one view element. Changes to the model element will be propagated to the view and vice versa.

[3]http://www.eclipse.org/gef

For example, the TAMo model contains a Task element which has a graphical representation, e.g. its view. Both of them are linked by a Task controller that is able to react on user interactions with Tasks view or changes to the Task model element.

Naturally, a controller is the most complex part in the MVC paradigm as it is responsible to keep all parts in sync. The controller layer of GEF is divided into two general components: *EditParts* and *EditPolicies*. An EditPart is always linked to one kind of view as well as one kind of model element and it handles the creation of views, of EditParts for child views and connections between view elements. Furthermore, they are responsible for refreshing the view in case of a model change. During the creation of an editor window, a concrete EditPartFactory is assigned; thus implementing a single model with different visual representations can be accomplished by providing several EditPartFactories with corresponding EditParts. While EditParts are responsible for the creation of elements, the second part of a GEF controller, EditPolicies, cover the editing aspect of the editor like moving, scaling or deleting views. Every kind of action is represented by a single EditPolicy and all desired Policies must be added to an EditPart element to allow for using these actions with the EditPart-view-model triple. To propagate changes, Policies use the Command pattern [Gamma et al., 1994] which enables the whole framework to provide nearly limitless Undo and Redo functionality.

View A GEF View is a graphical representation of a single model element. The view itself does not contain any data or logic. All information that is necessary to draw the view is received from the corresponding controller. Similar to the other two MVC components in the GEF framework, the view hierarchy corresponds to the model respectively the EditPart hierarchy. View elements are created alongside the corresponding EditPart and its graphical representation relies heavily on the *Draw2D* framework which is described in section 7.1.3.

7.1.3 Draw2D

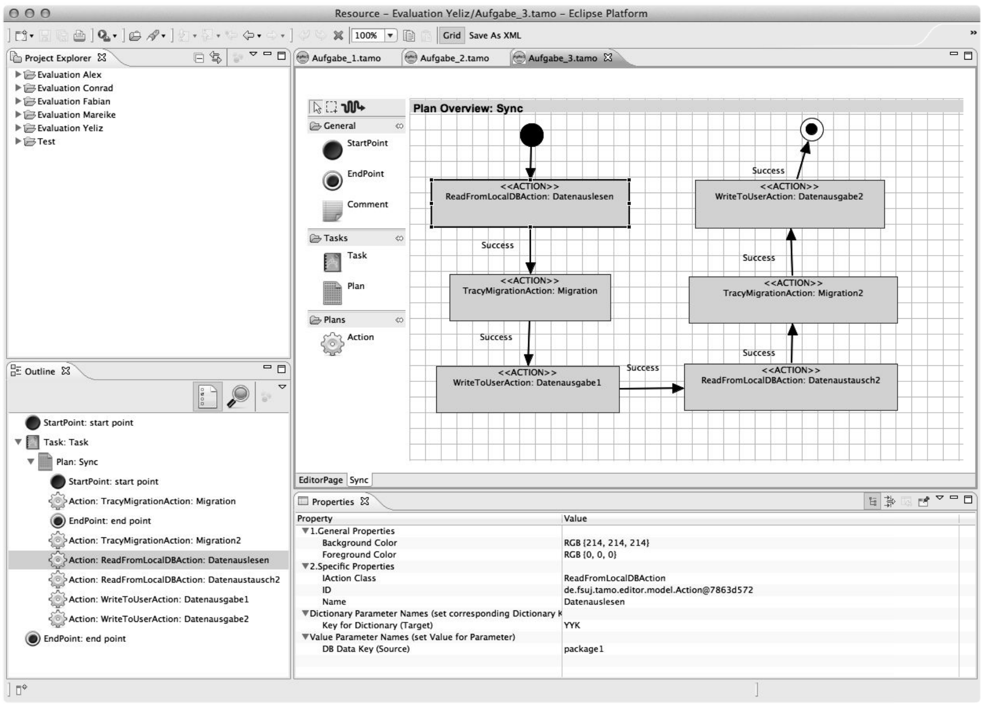
The GEF plugin itself uses the *Draw2D*[4] plugin for drawing primitive shapes like rectangles, connections or simple labels. In contrast to simple drawing libraries, Draw2D features the notation of a *Figure* and a complex graphic is represented by a hierarchy of Figures. Similar to a classic Composite Pattern [Gamma et al., 1994], every Figure

[4]http://www.eclipse.org/gef/draw2d

may contain other Figures and is responsible to draw itself and its children. Thus, by combining drawing primitives like rectangles, lines and labels, creating and handling complex graphical representations of a model is straightforward.

7.2 Editor Elements and Functionality

After having described several general properties of Eclipse-based editors, we will have a detailed look at the TAMo editor and its parts in this section. The TAMo editor showing a Plan model is presented in figure 7.1.


Resource – Evaluation Yeliz/Aufgabe_3.tamo – Eclipse Platform
Grid
Save As XML
Project Explorer
Evaluation Alex
Evaluation Conrad
Evaluation Fabian
Evaluation Mareike
Evaluation Yeliz
Test
Aufgabe_1.tamo
Aufgabe_2.tamo
Aufgabe_3.tamo
General
StartPoint
EndPoint
Comment
Tasks
Task
Plan
Plans
Action
Plan Overview: Sync
<<ACTION>> ReadFromLocalDBAction: Datenauslesen
Success
<<ACTION>> TracyMigrationAction: Migration
Success
<<ACTION>> WriteToUserAction: Datenausgabe1
Success
<<ACTION>> ReadFromLocalDBAction: Datenaustausch2
Success
<<ACTION>> TracyMigrationAction: Migration2
Success
<<ACTION>> WriteToUserAction: Datenausgabe2
Success
EditorPage
Sync
Outline
Properties

StartPoint: start point
Task: Task
Plan: Sync
StartPoint: start point
Action: TracyMigrationAction: Migration
EndPoint: end point
Action: TracyMigrationAction: Migration2
Action: ReadFromLocalDBAction: Datenauslesen
Action: ReadFromLocalDBAction: Datenaustausch2
Action: WriteToUserAction: Datenausgabe1
Action: WriteToUserAction: Datenausgabe2
EndPoint: end point

Property	Value
1.General Properties	
Background Color	RGB {214, 214, 214}
Foreground Color	RGB {0, 0, 0}
2.Specific Properties	
IAction Class	ReadFromLocalDBAction
ID	de.fsuj.tamo.editor.model.Action@7863d572
Name	Datenauslesen
Dictionary Parameter Names (set corresponding Dictionary K	
Key for Dictionary (Target)	YYK
Value Parameter Names (set Value for Parameter)	
DB Data Key (Source)	package1

Figure 7.1: TAMo Plan Editor

7.2.1 Model

As mentioned in section 7.1.2, we started with the model of the TAMo engine and extended it to incorporate all the requirements imposed by the Eclipse respectively GEF framework. To propagate different types of functionality to several or all model objects, a hierarchy with four layers was created. Upper layers provide basic functions like allowing for the connection of model elements and giving them a location on a two dimensional plane. Lower layers offer more specialized features like the publication of attributes to a Properties view or the ability to handle a parent-child relationship. The leaf elements in this hierarchy represent the actual model elements that are manipulated in the editor. In the following, we will outline each of the classes which represent the different layers in detail.

Class AbstractElement The base class of the whole hierarchy is *AbstractElement*. It provides basic features to all elements of the model like the ability to clone an element by implementing the *java.lang.Clonable* interface – a requirement to support *copy&paste*. To use the save and load functionality offered by the Eclipse framework, the *java.io.Serializable* interface is implemented, too. Beside that, it incorporates the Observer pattern [Gamma et al., 1994] to enable all model elements to react on changes propagated by their corresponding controller. AbstractElement is defined as an *abstract* Java class so no instances can be created. The only model element that is a direct subclass of AbstractElement is the *Connection* class which is only responsible to connect other model elements with each other.

Class AbstractChartElement All attributes, which are shared by all but the Connection model element, are established in the *AbstractChartElement* class. Notably, these include an element name, the position of the element in the plane, the dimension of the element, an optional parent element and two lists for incoming respectively outgoing connections. The complete handling of adding and removing connections can also be found here. Furthermore, *AbstractChartElement* provides all means to publish an element's properties to the *Properties View* and handle the changes which occur there by implementing the *IPropertySource* interface. Direct subclasses are *Action*, *StartPoint*, *EndPoint* and *Comment*.

Class AbstractContainerElement The class *AbstractContainerElement* extends *AbstractChartElement* and adds support to handle parent-child relationships. For example, a *Plan* element as parent contains a number of *Action* elements as children. Direct subclasses are *TaskOverview*, *Task* and *Plan*.

Concrete Model Element Classes The leaf elements in the model hierarchy represent the only elements that can be instantiated and used in the editor. Based on their common parent classes, each of them offers similar properties that can be edited directly using the editor or by changing values through the *Properties View*.

7.2.2 Action Class Loading

A special kind of model elements are *Actions*. Due to the fact that each concrete Action is implemented by its own class in the TAMo engine, the Action element offers an additional property to select the concrete Action class which is represented by it. After such a selection, the editor creates an instance of this concrete Action class and accesses the property methods described in section 6.2.4 to acquire the concrete set of dictionary parameters and outgoing connection types. This information is afterwards added to the *Properties View* of the selected Action.

To achieve a loose coupling between the editor and the concrete action classes, the editor only refers to the *IAction* interface which all actions must implement. During runtime of the editor, the user can select a folder or JAR file that contains action class files. Out of this container, any files with a name that ends with *Action* is added to the list of available actions. Thus, there is no need to adapt the editor in any way when new action classes have been implemented. All new classes with their unique set of parameters and connections are fully support right away.

7.2.3 Editor

A single script for a new agent respectively an agents tasks is represented by a single file in the *Project Explorer* in Eclipse. When double clicking a TAMo file, a *Multipage Editor* window opens. On the left side of the editor window appears the so-called *Palette* which offers all actions and tools the user can utilize to create an agent. Namely, the Palette includes a Selection tool and several tools to create all model elements. When first opened, the editor will show the Task overview. The Task overview represents the first layer of the TAMo model and allows for the definition of Tasks and their interconnections. Furthermore, the user is able to add Plans to a Task. By double clicking a Plan element, an additional page is opened in the editor window revealing the editing view for this particular Plan. Combined, all Plan editing pages make up the second layer of the TAMo model. In the Plan overview, the user can combine *Actions* to create the flow of execution. Common UI metaphors like *drag&drop* or *cut&paste* are supported.

7.2.4 Properties View

Mentioned before, the *Properties View* acts as a complement to the editor windows by providing context aware configuration options for the currently selected element. The

Properties View will acquire its contents from the selected editor element if this element implements the *IPropertySource* interface, which all TAMo model elements do.

7.2.5 Outline View

The outline view is a kind of supporting view to make the usage of the editor faster and more convenient. It offers two different modes – a bird eye view on the complete diagram the user is working on and a hierarchy list which represents all the model elements in their corresponding relationship. While the first mode makes navigation in large diagrams easy – simply by moving the bounding box that represents the current viewport – the second mode offers a fast way to find and select a desired model element. See figures 7.2 and 7.3 for examples of both modes.

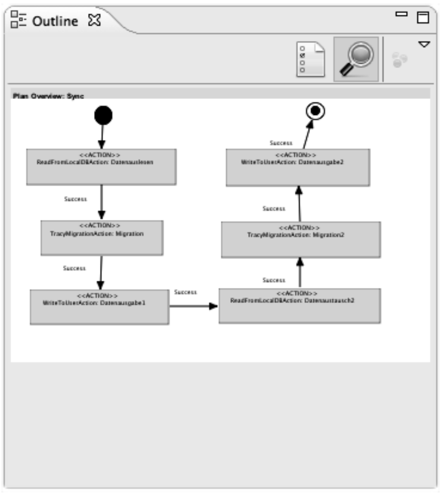
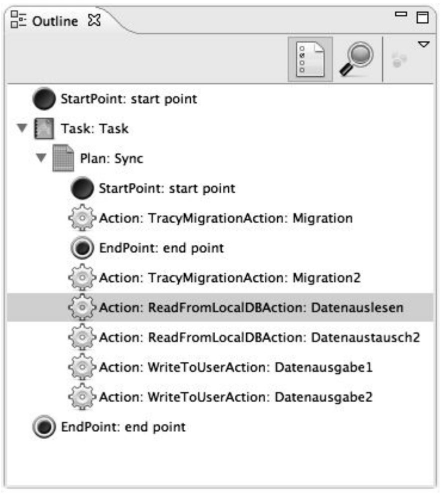

Outline
Plan Overview: Sync
<<ACTION>>
ReadFromLocalDBAction: Datenauslesen
Success
<<ACTION>>
TracyMigrationAction: Migration
Success
<<ACTION>>
WriteToUserAction: Datenausgabe1
Success
<<ACTION>>
ReadFromLocalDBAction: Datenaustausch2
Success
<<ACTION>>
TracyMigrationAction: Migration2
Success
<<ACTION>>
WriteToUserAction: Datenausgabe2
Success

Figure 7.2: Editor Outline View in Overview Mode



Outline

- StartPoint: start point
- Task: Task
  - Plan: Sync
    - StartPoint: start point
    - Action: TracyMigrationAction: Migration
    - EndPoint: end point
    - Action: TracyMigrationAction: Migration2
    - Action: ReadFromLocalDBAction: Datenauslesen
    - Action: ReadFromLocalDBAction: Datenaustausch2
    - Action: WriteToUserAction: Datenausgabe1
    - Action: WriteToUserAction: Datenausgabe2
- EndPoint: end point

Figure 7.3: Editor Outline View in List Mode

7.3 Linking Engine and Editor

After having created nearly all of the above described parts, the last remaining question was *How do we link the editor model to the engine?*. The underlying model structure of the editor is nearly identical to the one used by the engine; but not the same. So, simply serializing Java objects was not an option. We decided to introduce an intermediate XML format for transferring the created execution script between these two parts of TAMo. Creating an XML representation of an object hierarchy can be accomplished easily by using the JAXB[5] framework. JAXB is shipped with the Java SDK and allows for binding of Java objects to XML and vice versa. It even allows for the creation of Java classes using an XML schema description and the other way around. The configuration of such a binding is performed be adding code annotations to the class files.

Having a valid XML description of an execution script provides several advantages. First, import it into the TAMo engine was very straightforward by using JAXB again. Second, one can imaging the usage of such a description in other programs or tools, e.g. by running it through an optimization process or a test suite.

Beside exchanging the created execution flows, editor and engine are link on a additional level by using the same class files that represent the available *Actions*. As described in section 7.2.2, the editors only reference to the engine model is the *IAction* interface which defined all methods needed to introspect an *Action*. During runtime, the editor is able to load JAR files or traverse file system folders for new *Actions*, thus making them available to editor users as soon as the programmer has implemented them. Furthermore, this loose coupling allows for the distribution of the editor as a standalone program in a precompiled package. There is no need to recompile and redistribute the editor when new *Actions* become available.

7.4 Extensions

In its current state, the editor offers all means necessary to create execution scripts for the TAMo engine in a fast and easy way. However, there is always room for improvement and we will outline several ideas which come to mind.

First of all, we could improve the process how *Actions* are added to a *Plan* and linked with each other. As a first step, we could for example aid the script designer by suggesting follow-up *Actions* based on the values a previous *Action* has produced. The same applies for linking *Tasks* with each other or adding *Plans* to a *Task*. This would require some sort of semantic descriptions for those parameter to allow for a runtime evaluation and selection process. Having such a description would also allow for the runtime adaption during plan execution. Such a feature would move TAMo into the direction of state-based, planning agents similar to the BDI model.

A second improvement to the editor would be the integration of it into the *Wai Lin* Administration UI thereby allowing for the creation of agents and controlling of agencies under one hood. With this integration, it should be possible for the user to

[5] http://jaxb.java.net

send the final execution script directly to a remote agency and start an agent with it. An even better feature would be the realtime tracking of the execution of a TAMo script thereby highlighting the currently executed parts in the editor window and displaying the current contents of the data storage alongside with some data about the agent itself. To achieve such a functionality, delegates for Tasks, Plans and Actions could be used.

Further improving the editor itself could involve some sort of validity check, like issuing warnings for *Actions* with no incoming connections as these will never be executed.

Chapter 8

Evaluation

During the last steps of the implementation of the TAMo engine and associated editor we started to prepare two evaluations to test and hopefully reinforce our claims concerning the developed system. This chapter will introduce those evaluations, outline the complete process that was conducted and discuss and judge the achieved results.

8.1 Evaluation Types

Due to the varying type of our theses, we have chosen different approaches to evaluate each of them. First, one evaluation aimed at comparing the development and usage of agents using the traditional Tracy 2 agent programming to the new TAMo agent framework. Here, we wanted to measure not only the runtime performance of agents that have been developed using either way but also the time required to get used to the tools and frameworks and the time to implement them. This first evaluation is described in detail in section 8.2.

The second evaluation was aimed at reinforcing the claim that non-experts would be able to create agents using the TAMo framework. Due to the rather soft nature of this claim, we conducted a qualitative evaluation with a five persons, who had no computer science or programming experience at all. Non of them had previous experiences with agent systems or agent programming; moreover, half of them did not know the whole concept at all. This second evaluation is discussed in section 8.3.

8.2 Expert Evaluation

As mentioned in the first section, the first conducted evaluation aimed at comparing development efforts and runtime performance of Tracy 2 and TAMo agents. In the following, we will outline the structure and goals, describe the conceptual scenario which was implemented as well as the steps that have been carried out during this process. Finally, we will present and discuss the results.

8.2.1 Structure and Goals

The evaluation was divided into the two different paths with each one consisting of two phases; namely the implementation of an agent based application and the runtime measurements. To prevent the results from being biased, we employed two computer science students with no prior knowledge of agent system development. Each student was responsible for one path and they had to work independently. Both were given the same scenario and description of the agent which had to be developed. We used a moderately complex real world scenario similar to the conference booking agent described in section 4.2. The scenario will be outlined shortly in the next section.

As a prerequisite task, both students had to install and configure the required Tracy 2 agent system and, in case of one student, the TAMo framework composed of engine and editor. Both students worked with the same version of the Tracy 2 agent system and, while implementation was done on different machines, all runtime measurements were performed on the same infrastructure, e.g. hardware, network, operating systems, and Tracy respectively TAMo versions.

For the scenario, several required third party services were implemented, e.g. a service offering travel options like hotels and flights. These services where created before the evaluation and the cost for implementing them was not included in the evaluation figures.

Some of the key figures that we wanted to track during the implementation phase were somewhat hard to capture respectively express in numbers. However, in such cases, we will describe and justify the results in the best possible way. During the implementation phase, we tracked the following key figures:

Learning Curve One of our claims is that using TAMo, it is much easier for developers and even end users to create agents – even without previous knowledge. Thus, the time someone needs to understand the given tools and frameworks and use them to create value will be considered as *Learning Curve*. However, this value is a bit hard to grab because one has to define a moment, where the learning is completed – and such a point is probably not specifiable, as everybody will improve its knowledge about tools while using them. Therefore, we defined this moment as *The user is confident that she can use the tools and frameworks to create something valuable and non-trivial.* In our case, this meant the point in time where the students stopped to experiment with the tools and started to implement the scenario and we captured the time up to this point.

Implementation Time The time it took to implement the scenario using either technique. Of all tracked figures this one is the easiest to measure.

Maintenance Effort A rather soft figure, which denoted the perceived effort that was required to alter and adapt an existing agent respectively TAMo script to comply to changed requirements. The important aspect of this figure was how fast can someone, who has general knowledge of the technology but did not implement the previous version, perform changes.

Reuse Prospects Again, a soft figure. It captured the prospects of reusing existing parts of an agent or TAMo script in future implementations. It depended mainly on the coupling between different parts and the generality of a parts function.

For the runtime measurements, probes had been added to the created agents to track their runtime performance in a very fine-grained way. Naturally, we were most interested in the overall running times as well as overall and single migration times. Other values, like the times spend on different agencies or calculating results, have been captured, too.

8.2.2 Scenario

the scenario used during this evaluation was an adapted version of the scenario presented in section 4.2. We made some slight adjustments to keep the overall efforts of the evaluation within acceptable bounds. We decided to use a single travel service instead of different services for accommodation, flight and car booking because accessing different services and evaluating their results would be very similar from the modeling respectively programming point of view and we would not gain any further insights by doing the same thing several times. Due to the fact that by using just a single service for all travel informations, we lost the final evaluation and matching task the agent had to fulfill. To compensate for this loss, we introduced optinal travel companions that would alter the travel requirements and change the way the agent would select a journey offering. To summarize, the used scenario was as follows. The developed agent was considered as a personal assistant to a researcher. Its task was to overlook the researchers calendar and, in case a new appointment, like an offsite conference, was added, to collect all required information for the University accountancy to book the journey. After starting its search, the agent contacts the personal assistants from other researchers to determine if these researchers will attend the conference, too. If so, it would widen the travel options, like sharing a car, or the agent could request discounts. Afterwards, it would acquire a list of travel services it could use to gather the required information. Finally, after having visited those travel services, the agent would calculate the best offer respectively the best combination of offers and, before returning to its owner's machine and deliver them to the University accountancy.

The scenario contained a number of hosts respectively agencies that fulfilled different tasks. Namely, these where

Scientist Workstation We had two machines running as the workstations of our scenarios scientists. On each machine, the agency hosted the scenario agent, which was waiting to start its *travel search and order* tour based on an external event, e.g. the scientist who added the conference to his calendar.

Travel Service Discovery This host offered a list of agencies which provided travel information, e.g. offering accommodations, flights or rentable cars. The agent visited this host at the beginning of its itinerary.

Travel Services Agencies that ran Travel Services provided offers according to the agents request like the overall price for a hotel room or the scheduled flight times. We used ten different Travel Service agencies/hosts during our evaluation.

8.2.3 Evaluation Execution

This section gives a complete summary of the evaluation process. It starts with a description of the general prerequisites that where necessary to start with the individual parts. Thereafter, the two implementation paths and the runtime measurements will be outlined.

8.2.3.1 Prerequisites

Before starting with the development of the agent, several preparations had to be made. First, Tracy 2 was installed on several machines and configured. To increase the number of available agencies, Tracy 2 was adapted so that several instances could run on the same machine. Furthermore, several small web services that mimicked the third party offerings where implemented as well as the personal assistant agent of the *other* researchers. Finally, to allow for the development of TAMo agents, the TAMo engine and editor where installed and configured for one of the two students. All in all, these preparation took roundabout 30 hours.

8.2.3.2 Tracy 2 Implementation Path

This track was targeted at the implementation of the scenario in Tracy 2 using only traditional Java programming. Here, a single class was created that contained the complete agents code. As mentioned in section 2.6.2, Tracy 2 agents are usually backed up by a state machine and so was the agent that was developed for the evaluation scenario. The different states were as follows:

State 1 The agent resides at the researchers machine and waits for an external event to trigger its tour. To keep things simple, we did not connect the agent to an actual calendar; the event was triggered by changing values in a file that was stored at the machine's hard drive.

State 2 The agent traverses the other researcher's machines and contacts the local agents to determine if one or more travel companions are available. The list of those machines was hard wired and not acquired during runtime.

State 3 This state describes the actions necessary to get a list of travel services. It involves a migration to an agency that hosts a designated service directory. From the local service, the agent receives a list of agencies which provide travel services like renting a car or booking a hotel.

State 4 In a roundtrip, the agent would traverse all travel service agencies and gather the different offers. To simplify things, each travel services provided the complete set of services: each one offered hotel, cars, flights and so on. So, at each travel service agency, the agent received a grand total price for the journey.

State 5 In the final state, the agent migrated back to the University and deliver the best offer to the local accountancy to execute the order. Afterwards, it returned to its owner's machine and waited for upcoming external events.

The development of this agent using traditional Tracy 2 methods took roundabout 30 hours.

8.2.3.3 TAMo Implementation Path

Creating the TAMo agent version required more steps than the other version because the specific *Action* classes for our scenario had to be created. We will first outline this implementation part and afterwards look at the script creation using the editor.

Due to the fact that all our services used in the scenario just mimicked real web services and used rather proprietary interfaces, several Action classes were created to access them.

WatchCalendar The first action was rather simple – it just checked the file that mimics the researcher's calendar in regular intervals. If a change occured, this action was completed and execution moved on.

SearchTravelCompanions The second action interacted with the other researcher's agents to check if they will visit the same conference. This action used the Tracy Message Plugin to communicate with these agents.

AcquireTravelServicesList This action accessed the travel service directory to get a list of available travel service agencies. The list consisted of a number of host respectively agency addresses which constituted the upcoming itinerary.

AccessTravelService For each travel service the agent visited, this action was executed to acquire an offer.

SubmitResults Finally, the last created action selected the best offer and submitted it to the accountancy.

SendEmail If the migration to the owner's host at the end of the tour was not successful, the agent sent the results via email.

Creating these *Action* classes required roughly 27 hours.

The second step was the creation of the TAMo execution script using the newly created actions as well as several available action classes for migration, loops or dictionary access. The script contained a single Tasks with two plans – the first one straightforward without much error handling whereas the second one did cope with failed migrations. For example, a failed migration to a travel service did not break the execution flow; instead the agent just continued the loop and tried to migrate to the next travel service. See figures 8.1 and 8.2. Both plans were created in 4 hours using the TAMo editor.

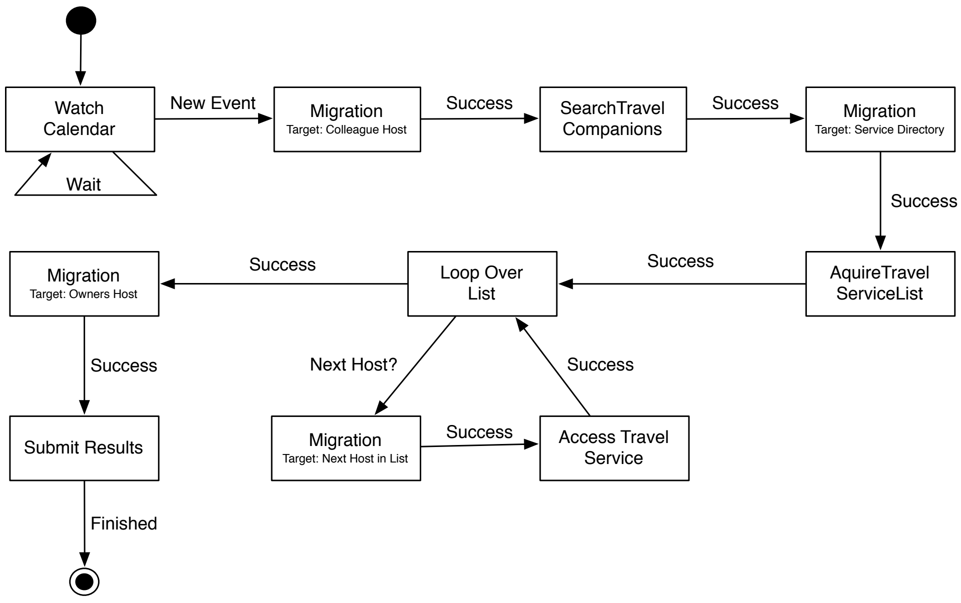


Figure 8.1: First TAMo Script for Evaluation

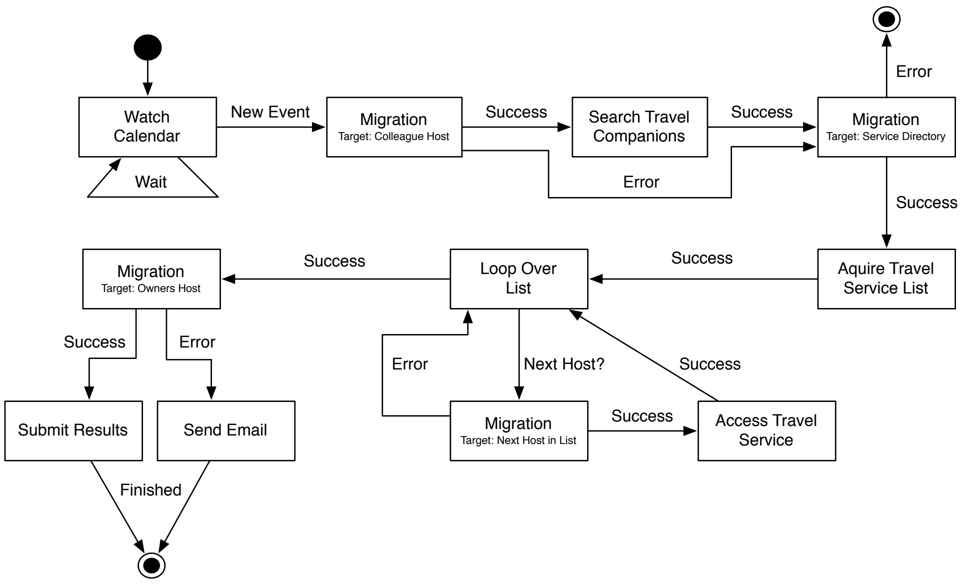


Figure 8.2: Second TAMo Script for Evaluation

8.2.3.4 Runtime Measurements

For our runtime measurements, we used two researcher agencies; one with our scenario agent and a second one running the agent that answered travel companion requests. A third agency was used to host the travel service directory. Ten additional agencies functioned as travel services. Delivering the final result took place at the scenario agent's home agency, e.g. the researcher's machine. Each of the two agents was executed 100 times and the top 2.5 percent of the fastest and slowest times where removed. The mean value was afterwards calculated from the remaining results. All times have been measured in nanoseconds.

To ease the analysis of the migration times, the complete code for the traditional agent as well as for the TAMo agent was already deployed at all agencies. Thus, migration did not involve transferring code but only the packaging and transmission of state and data information. This enabled us to better compare the migration times because the amount of code between the two paths was very different. It could be argued, that such a constraint limits the significance of the results with regard to real world scenarios. However, in [Kern et al., 2004; Kern and Braun, 2006] we conducted experiments to increase the migration performance with special consideration of an agent's code. Techniques like altering an agents Java code on Bytecode level could be used to create equal code bases for both agents which would remove the impact of code transmission completely. Therefore, intentionally avoiding the effect of code size during the evaluation did not limit the relevance of the results.

8.2.4 Results

This section delivers the complete measurements of the evaluation. Starting with the development times, table 8.1 displays the efforts that were necessary to create both agent versions. At a first glance, both paths seem to be fairly equal. However, it must be noted that the efforts for the TAMo agent include the creation of all proprietary *Action* classes which were required in our test scenario. The creation of the execution script itself was a magnitude faster than writing the agent from scratch. Assuming that over time, the number of generic *Actions* which can be used in a variety of cases increases significantly, building a TAMo based agent should be quite fast compared to the traditional methods. Moreover, even if most of the actions must be written from scratch, the concept of small, single-focused code fragments should lead to better code quality, easier maintenance and easier reuse. Comparing only the development times for the traditional agent and the action classes shows that creating the latter ones was a bit faster – probably because the code does not involve the complete state machine handling which is all hidden inside the TAMo framework. Thus, the developer can concentrate on the core functionality without writing tedious boilerplate code.

Table 8.2 shows the runtime measurements of the two agent versions. We have tried to match the states of the traditional agent with the Action classes used by the TAMo agent to ease the comparison. Due to the rather small amount of actual work, both agents spent most of their running time during migrations. As can be seen from these values, the TAMo agent was roughly two and a half times slower than the traditional agent – 1.9 versus 0.7 seconds. This difference is solely produced by agent migration as a comparison of the remaining time values shows; with 256 and 205 ms, both agents performed the actual work nearly equally fast and the difference can certainly be attributed to implementation differences.

	Time in Hours
Tracy 2 Agent	30
TAMo Actions	24
TAMo Script	4

Table 8.1: Implementation Efforts

	Tracy 2	**TAMo**
Migration	433	1779
Work	256	205
Total	**689**	**1984**

Table 8.2: Runtime Results (in ms)

So, the question, why did the TAMo agent perform so bad during its migrations remains. As stated in section 8.2.3.4, we did not transfer code during an agents trip so the bigger code base of the TAMo agent could not be the reason. However, during a migration, all objects that belong to an agent are serialized and transmitted. So, even removing the effect of the code base, the significantly higher amount of involved classes and thus object leads to significantly bigger migration times. The traditional Tracy 2 agent was just a single class and during a migration, a single instance of that class was serialized and transferred. Moreover, the state of that instance captured not only the agent's current status but also held all the data the agent had collected so far. In contrast, a TAMo agent is made up of several objects that are instances of the core TAMo framework classes like *Plan* as well as an instance for every *Action* the agent uses. Beside that, the data which a TAMo agent has collected during its itinerary is stored in an external dictionary which is serialized as well. To make things even worse, during its startup phase, a TAMo agent initializes all Tasks, Plans and Actions which are found in its execution script. So, regardless the fact if a specific plan is ever used, the agent will carry it around during the whole tour. In our evaluations, the TAMo agent consisted of roughly 25 class instances during runtime and this, compared to the single instance of the traditional agent, explains the huge difference for the migration times.

8.2.5 Discussion

The following paragraphs will discuss the achieved results of our evaluation. With a first look at the sheer numbers, the traditional Tracy 2 version of the agent seems to be the winner as there are nearly no differences in the development time and its runtime performance is way better. However, as stated in the beginning of this chapter, several figures we wanted to track during the evaluation can be hardly expressed in sheer numbers. Taking these figures like maintainability into consideration, this first impression changes significantly.

8.2.5.1 Development

Comparing the numbers, both agents could be created in roughly 30 hours and both of them where capable to fulfill the given tasks. It must be noted that the TAMo version involved a very high amount of ordinary programming to create the different *Action* classes required to access the proprietary travel services. Due to the lack of existing actions, our idea of TAMo, as a fast way to create agents from a large toolbox of existing actions, could not show its strength in this evaluation. If one imagines that these *Action* classes had existed, we could remove 27 hours of implementation time. In that case, the creation of a TAMo agent had been much faster – with just four instead of thirty hours. Even if one or two actions had to be implemented, creating them would have surely been faster than creating a whole agent.

But even if we do not remove the implementation time for all *Action* classes, the TAMo path has, in the same time, lead to much better results in terms of software quality, maintainability and ease of reuse. The Tracy 2 agent is a single, large Java class that performs a variety of tasks with a custom state machine to switch between them. Such an implementation is usually hard to understand and therefore hard to maintain. Moreover, it is probably difficult to reuse any of the written code because the different parts are likely entangled in a strong way, cannot be removed easily and used in another context. In contrast, the TAMo implementation efforts have lead to five, single-purpose actions, that can be used by any TAMo script. They define a clear interface and anyone, who has seen or programmed a TAMo action, should be able to maintain and adapt these classes. Using the editor, one can even make changes to the agent without touching any code – something which is impossible to achieve with the

Tracy 2 agent. Last but not least, any written action increases the number of available actions and thus the powerfulness of the complete framework.

8.2.5.2 Runtime

With the development result showing equal efforts for the creation of both agent versions, the runtime results depict a clear advantage of the Tracy 2 agent which was 2.5 times faster. In section 8.2.4, we analyzed these differences and found several reasons for the worse performance of the TAMo agent. Now, we will describe several techniques to address them, decrease their impact and thus narrow the performance gap.

Essentially, we identified two aspects that lead to the comparatively large running times. First, the TAMo agent is made up of a lot more classes, and thus objects, than the traditional agent. Beside the fact that we removed the impact of class file transmission during migration, the sheer number of objects which are transferred during each hop increase the migration times significantly. There are several options available on how we could decrease the number of classes used by the TAMo agent:

Bytecode Modifications As mentioned before, the modification of Java Bytecode can be used to alter the size and structure of an agents code base. In our past research, we used this technique to create a more fine-grained code base for agents which, similar to the Tracy 2 agent in our evaluation, consist of only one or a handful of classes. By altering the Bytecode and moving parts of the code into new classes, we could achieve a much better migration performance [Kern et al., 2004; Kern and Braun, 2006]. We are confident, that a similar approach can be used here – only in the reverse direction. By joining different classes into a single one, the fragmentation of an agents code can be reduced.

However, our past efforts to increase the granularity of an agents code base where clearly directed at utilizing the power of the Kalong migration engine [Braun and Rossak, 2004] and thus improving the migration performance of mobile agents. So, moving in the reverse direction is a bit odd. One could even question the representativity of Tracy 2 agent used in this evaluation. How likely is it, that more complex, real-world agents consist of only a single class? Maybe we should not aim at creating single class agents using Bytecode modifications but instead try to optimize the fragmentation of an agent's code based on various conditions, like network throughput, number of migrations and hosts visited, ratio between data and code size or timing constraints imposed by the agents owner.

Code and Mirror Servers As described in section 2.5.1, it could be shown that the usage of *Code* and *Mirror Servers* could greatly increase the migration performance of mobile agents. Thus, the TAMo agent could store unused classes and objects to decrease its overall migration size. To hide these aspects from the agent designer respectively non-expert, the TAMo framework itself could internally optimize an agent's size during runtime. This would free an agent's execution scripts from such task-independent parts and would allow for all scripts to benefit from these optimizations. Such an automated usage of code or mirror servers could try to store unused entries of the dictionary. Due to the fact that every action exposes its used dictionary entries, it would be easy for the framework to decide which entries will be required or not by analyzing the currently executed plan and upcoming actions. During the same analysis, the framework could determine classes and instances which are not required right now or which have been used before and will never be required again. These parts of an agent's code and state could be moved to a mirror server as well.

Framework/Engine Modifications Beside the other two options which are essentially extensions, the third option aims at altering the new framework directly. For example, we could try to alter the general structure of the base classes to reduce their overall number. For example, moving the handling for *Plans* into the *Task* and further add the complete dictionary. There are surely several options to reduce the footprint of the core classes. However, we are somewhat reluctant to decrease the quality of the implementation out of pure performance reasons – even more when there are other options available.

The second aspect that causes the higher migration times is the manner in which a TAMo agent is initialized. The process itself is straightforward, but it imposes a huge burden on the running times. Currently, the XML file that was exported from the editor is parsed in a single step and all contained elements are created and initialized at the same time during the agent's startup phase. This means that, regardless of usage, every *Task*, *Plan* and *Action* that is contained in the script is created upfront. The script for the example agent contained a single task with two plans and, if everything ran as expected the second plan was never used. However, in the current implementation, all elements, e.g. actions and interrelationships, are created during the startup phase. Thus, the agent carried around a *Plan* and all its elements during its whole runtime even if those parts were never used.

Having identified this drawback, we will alter the complete process. We aim at creating required elements on demand and thus reduce the number of transferred objects significantly. This change involves an adaptation to the XML structure which, in its current format, cannot easily describe a *Plan* as we use it. For example, the XML file contains all *Actions* and their interrelationships but to discover loops and similar constructs, one has to parse and analyze the XML file as a whole. Therefore, we aim at adapting the XML as well as the process to initialize the agent itself. The overall goal is that we can create *Plan* and *Action* class instances on demand. Thus, we will have to establish a structure, which describes a *Plan* but where concrete parts are only instantiated, if needed. Creating this structure will probably involve adaptations to the core agent as well as the core classes that make up the TAMo framework. However, removing respectively avoiding the instantiation of unused parts of an agent script will, without doubt, reduce migration times.

To summarize, we could identify and qualify the two main aspects that lead to the rather poor runtime respectively migration performance of the TAMo agent when compared to the traditional implementation. Several options to eliminate these reasons have been presented and discusses and with regard to all other aspects, like maintainability and reusability, that make up a framework, we are confident to improve our TAMo engine to stand up against existing techniques – not only with respect to ease of use but also performance-wise. Overall, we could show that the gap between the traditional and the TAMo agent is identifiable and addressable and that there are various options to narrow this gap.

8.3 Non-Expert Evaluation

The second evaluation aimed at verifying our claim that TAMo has a flat learning curve and is usable by non-experts. This claim is rather difficult to check as it is hard to express a *flat learning curve* and *usable* in numbers. We therefore conducted a qualitative study with non-expert participants which had to create several agents using the TAMo framework and monitored their performance during the evaluation.

8.3.1 Participants

For this evaluation, we needed participants with no background in agent technology and programming in general. Thus, we selected five participants with an education in social science or business administration. All of them had general knowledge regarding the usage of computers, but non of them had any prior experiences with software development, development tools like Eclipse or agent software engineering.

8.3.2 Evaluation Execution

The evaluation started with an introduction into the concepts of mobile software agents as well as the core ideas behind the TAMo framework. Afterwards, the graphical editor was demonstrated and the agents, that should be developed during the evaluation, were presented. All in all, the briefing took roundabout 15 minutes.

During the evaluation, each participant had to create three different agents. The first one was a simple *Hello World* agent that would just greet the user. It was meant to teach the participant the general handling of the graphical editor as well as the usage of model elements like *Tasks*, *Plans* and *Actions*. The second was an extension to the first agent and involved the migration to another agency. At the remote agency, the agent should greet the user again. This task was meant to familiarize the participants with the concept of agent migration and how it is modeled. The third, most complex agent should synchronize data between two agencies. Here, the participants should learn how to use the global dictionary that is associated with every agent and exchange information with local databases. Figures 8.3, 8.4 and 8.5 show the final plans for all three agents. All necessary actions to build those agents were given.

8.3.3 Results

During the evaluation, the biggest hurdle for all participants was to get used to formal thinking any computer scientist is familiar with. Connecting Actions in a meaningful order by matching input and output proved to be more difficult than understanding the concepts of agents or agent migration. Beside that, all participants had no problems using the graphical editor and some of them came up with useful suggestions to improve the editor's usablitiy in several ways. For example, being forced to give any Action used in a Plan a speaking name would greatly help to distinguish different Actions of the same kind. Furthermore, one participant suggested that it would help to give all Actions of the same kind the same color, e.g. all Migration Actions could be blue, but differnt kinds should have different colors, e.g. blue for Migration Actions blue and green for Loop Actions. Currently, the editor allows for setting the background color of Action elements by hand but defining default colors for different kinds of Actions would be a helpful addition.

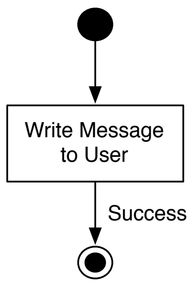


Figure 8.3: Script for Agent 1

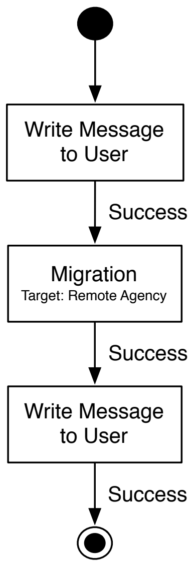


Figure 8.4: Script for Agent 2

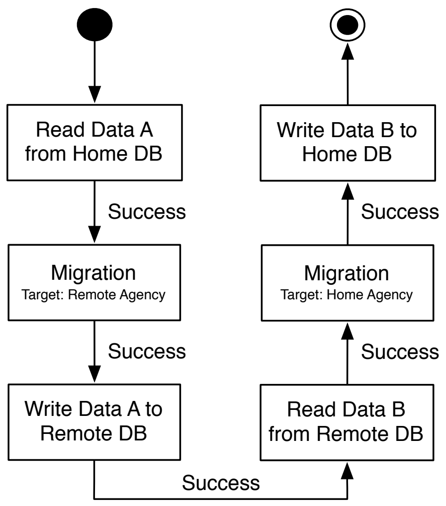


Figure 8.5: Script for Agent 3

Summarizing, all participants were able to successfully create the three agents which have been verified by a runtime test afterwards. Developing these agents took between 25 and 45 minutes. Thus, including the introduction, getting started with TAMo took not longer than one hour. As stated before, the number of participants does not qualify this study as statistically significant. However, we consider the results as very promising and reassuring.

Chapter 9

Discussion and Future Work

In this chapter, we will critically revise the conducted work, recount our aims and thesis and compare them to the achieved results. We will start by revisiting our theses and and examine if the results reflect them. Afterwards, we will present some ideas for improvements of the presented work.

9.1 Theses Revisited

Thesis 1: It is possible to develop an agent model and a runtime environment to execute predefined plans by using a well known, industry proven programming language like Java

The first theses can be verified very easily because it is a fact that the complete TAMo framework is implemented in Java and allows for the creation of new functionality in terms of actions by using the Java programming language and common IDEs. Furthermore, the graphical editor was implemented on top of Eclipse, the most widely used Java IDE. The editor can be used as a plugin in an existing Eclipse instance but also as a standalone application. The first option is targeted at programmers that already use Eclipse and want to create and test TAMo actions. The second option is meant for non-experts who just want to create TAMo agent scripts based on an available set of actions. Thus, we believe that the TAMo framework fits very well into the general environment and toolset of many developers as well as providing an easy and fast entry into the framework for non-experts.

Thesis 2: It is possible to create an agent development environment that is easier to use than any of the currently available ones

The second thesis is rather hard to verify. How does one define *easier*? However, the similar frameworks, which where already presented in section 2.4.4 and 5.3.1, are clearly targeted at agent experts and require a profound knowledge of agent systems. However, we could show that our approach to separate the programming of actions and the definition of agent scripts enabled non-expert users with no background knowledge to create agents. We are aware that the number of participants in our non-expert evaluation was far to small to refer to the results as a proof. But we believe that these results reinforce our claim that TAMo is very easy to use and probably the easiest agent development framework present today. Furthermore, with the concept of atomic, reusable actions, we could introducing a more sophisticated level of reuse and maintainability.

Theses 3: The proposed framework will allow for faster development and execution cycles as well as provide better software quality

Creating and changing agents using the graphical editor is fast and, due to the fact that all actions are precompiled, executing and testing does not require a compilation step. The user simply saves the script and starts an instance of the *CoreAgent* to execute it. And, using the editor to exchange actions or adapt their order, altering an agent is easier and faster than doing the same at code level. Given the fact that, with a large set of available actions, agent creation will primarily performed using the editor, this speedup in development time is significant.

For the creation and adaptation of actions, compilation is still necessary. However, it is usually much faster to compile a single action than a complete agent. Moreover, because the action will show up in the graphical editor immediately after compilation, running and testing them is very fast.

9.2 Future Work

Looking at the TAMo framework in its current installment, several useful extensions come to mind. First, the process of selecting one of the possible numerous plans of a task for execution could be made more sophisticated. As stated in chapter 6, the current plan selector implementation simply selects the next plan in the given list of plans. This process could be enhanced by introducing some kind of learning behavior that incorporates statistics of past executions as well as information on the environment during these executions. For example, services used in a plan could be overwhelmed by requests during certain times of the day or week. Having such information, the selection process could skip such a plan during those times and rather select one that has proven to be more reliably. Or, considering the overall running time of a plan, in certain situations, a plan might just not be fast enough and knowing this in advance would prevent the agent to fail on the complete task by missing a critical timeframe. Further research could also focus on optimizing agent performance with regard to the communication strategy, e.g. remote access or agent migration, by selecting the most appropriate plan for execution.

A second extension could be the addition of pre- and postconditions to actions, plans and tasks. At best only to actions because conditions for plans and tasks could be derived from the contained actions. This addition would allow for two new functionalities. First, during the design process, the editor could support the user by suggesting compatible actions by matching post- and preconditions of two different actions. This would make script creation even more easier for non-experts, because connecting actions in a non-useful manner would be impossible. Beside the editor enhancement, the TAMo engine could be adapted to allow for more flexibility during runtime. For example, one could image a plan selector that is able to search for plans that would match the current task but that where not available during the design process of the currently executed script. Or the engine could adapt an executed plan by exchanging actions with new ones which have been developed after the creation of the script and that offer better results or performance.

Beside its usage as an agent execution engine, TAMo can be used as a standalone runtime for scripts. We would like to use it in various environments and applications and evaluate its applicability and usefulness. For example, it could be integrated into application server infrastructures to run reoccurring background maintenance tasks. It should even be possible to apply it in a web server to serve requests with TAMo scripts.

But the most desired extension to the TAMo framework are simply more actions to increase the flexibility and usefulness of the whole framework. Our hope is that, over time, the set of reusable actions reaches a size that allows for the creation of agents by only using the graphical editor and to employ a programmer for a new action only if required in a few border cases.

Chapter 10

Conclusion

Mobile agents are a fascinating concept to design distributed systems. The field has evolved from a mere scientific topic to a technology that is used in a variety of applications like sensor network management or cloud computing. Beside their usage for autonomic maintenance tasks, mobile agents have always been regarded as personal assistants to their human users. However, the utilization of such assistants is still reserved for agent experts because the currently available methods and tools to create and adapt mobile agents are far to complex and specialized to suite normal users and, sometimes, even experts.

The aim of this thesis was to establish an approachable method and easy to use tools to model, create and use mobile agents. The ultimate goal for all techniques and frameworks to be developed was simplicity to allow for simple access and rapid successes with this fascinating paradigm.

In a first step, we defined three different types of targeted users ranging from non-experts to experienced developers. Based on these roles, we developed the concept of agents that are created by combining a set of basic, atomic actions in a useful manner. The main idea behind this approach is the strict separation of creating code and defining what a single agent should actually do. Whereas creating code is clearly a duty for a programmer, specifying the actual task of an agent should be possible for non-experts.

Around this basic idea, we established an agent model named *TAMo* and a graphical notation to specify instances of TAMo. Whereas the first version of this model introduced the concept of a Task and a Plan layer to allow for structuring an agent's task as well as easier reuse of agent parts, it was still to complex. During the evolution into the second, final version of TAMo, many elements where dropped to even more embrace the idea of atomic actions. At its core, an agent consists of a set of Tasks that are backed up by Plans. Every Plan is thereby a collection of Actions that are connected in a useful manner.

During the next step, we developed an engine that is capable to execute TAMo model instances. Although we started with mobile agents in mind, the core engine is a standalone framework that can be integrated into any kind of application to run TAMo instances. Afterwards, this core engine was extended and integrated into the Tracy 2 agent toolkit as an agent execution environment. Due to its sleek set of dependencies, this integration was very straightforward and, thus, serves as a great reference on how easily the integration of the core engine into another application can be achieved.

To allow for the definition of TAMo models that are based on the established notation and that can be executed by the developed engine, we created a graphical editor. As with any other element of the TAMo framework, simplicity, ease of use and approachability where the primary goals. We selected the Eclipse workbench with its plugin concept as technological foundation out of two reasons. First, Eclipse is a widely used development environment among programmers. As such, it offers a large possible user base for TAMo. Second, it allows for creating standalone applications based on the core workbench and a set of plugins. We can, therefore, provide an application specifically for the creation of TAMo models to all non-expert users. The developed editor offers an easy mechanism to export TAMo models and execute them in an instance of the TAMo engine.

After the implementation of the TAMo framework and tools, the final part of this thesis where two evaluations to verify if our aims could be achieved. The first evaluation was targeted at experts and should compare agent development using TAMo versus the traditional approach available for the Tracy 2 agent toolkit, e.g. writing an agent's code by hand. The results showed that, given an extensive set of Actions to use, the creation of agents using TAMo can be significantly faster. Moreover, using established and tested Actions, the general code quality should be better whereas the graphical editor provides an easy and fast way to alter and adapt available agents. However, the runtime results showed some drawbacks of the current implementation of the core TAMo engine, for example much larger migration times when compared to traditional agents. By analyzing these areas of subpar performance, we could offer ideas for several enhancements that should yield significant improvements on the runtime performance of TAMo based agents.

The second evaluation aimed at testing the approachability and ease of use of the TAMo framework. After a short introduction into the general concepts and tools, a number of non-experts was given the task to build three different agents. Each participant was able to solve this task and successfully design mobile agents in less than an hour. We are aware that the number of participants was to small to consider these results as an evidence. However, we nevertheless believe that the results are a good indicator that TAMo offers a very approachable and easy to use way to create, adapt and use mobile agents and that it opens the door to this fascinating concept to a much larger audience.

Bibliography

Anurag Acharya, Mudumbai Ranganathan, and Joel Haskin Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130, London, UK, 1997.

Philip E. Agre and Stanley Joshua Rosenschein. Computational Theories of Interaction and Agency. The MIT Press, Cambridge, MA, USA, 1996.

Francesco Aiello, Giancarlo Fortino, Raffaele Gravina, and Antonio Guerrieri. A Java-Based Agent Platform for Programming Wireless Sensor Networks. *The Computer Journal - Special Focus on Agent Technologies for Sensor Networks*, 54(3):439–454, February 2011.

Raja Al-Jaljouli and Jemal Abawajy. Agents based e-commerce and securing exchanged information. pages 383–404. Pervasive Computing - Innovations in Intelligent Multimedia and Applications, London, 2010.

Gustavo Alonso and C Mohan. Workflow management systems: The next generation of distributed procesing tools. *Advanced Transaction Models and Architectures*, 1997.

Mercedes Amor, Lidia Fuentes, and Antonio Vallecillo. Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA. *Lecture Notes in Computer Science: Agent-Oriented Software Engineering V*, 3382(93-108), 2005.

Davide Ancona and Viviana Mascardi. Coo-BDI: Extending the BDI Model with Cooperativity. *Lecture Notes in Computer Science: Declarative Agent Languages And Technologies*, 2990:109–134, 2004.

Davide Ancona, Viviana Mascardi, Jomi F. Hübner, and Rafael H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange. In *AAMAS '04 Proceedings of the Third International Joint Conference on Autonomous Agents and*

Multiagent Systems, pages 696–705. AAMAS '04 Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004.

Kohei Arai and Lipur Sugiyanta. Energy Consumption in Ad Hoc Network With Agents Minimizing the Number of Hops and Maintaining Connectivity of Mobile Terminals Which Move from One to the Others. *International Journal of Computer Networks (IJCN)*, 3(2):71–86, May 2011.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Berkeley, February 2009.

Guillaume Autran and Xining Li. Implementing an Mobile Agent Platform for M-commerce. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, pages 40–46, Seattle, WA, July 2009. Dept. of Comput. & Inf. Sci., Univ. of Guelph, Guelph, ON, Canada.

Rocco Aversa, Beniamino Di Martino, and Nicola Mazzocca. MAGDA: A Software Environment for Mobile AGent based Distributed Applications. In *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.*, pages 332–338. Dipt. di Ingegneria dell' Informazione, Second Univ. of Naples, Italy, February 2003.

Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca, and Salvatore Venticinque. MAGDA: A Mobile Agent based Grid Architecture. *JOURNAL OF GRID COMPUTING*, 4(4):395–412, June 2006.

Rocco Aversa, Beniamino Di Martino, and Salvatore Venticinque. Integration of Mobile Agents Technology and Globus for Assisted Design and Automated Development of Grid Services. In *2009 International Conference on Computational Science and Engineering*, pages 118–125, Vancouver, Canada, August 2009. Dipt. di Ingegneria dell' Informazione, Second Univ. of Naples, Italy.

Rocco Aversa, Beniamino Di Martino, Massimiliano Rak, and Salvatore Venticinque. Cloud Agency: A Mobile Agent Based Cloud System. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 132–137, Krakow, Poland, February 2010. Dipt. di Ingegneria dell' Informazione, Second Univ. of Naples, Italy.

Yoram Bachrach, Ariel Parnes, Ariel D. Procaccia, and Jeffrey S Rosenschein. Gossip-based aggregation of trust in decentralized reputation systems. *Autonomous Agents and Multi-Agent Systems*, 19(2):153–172, 2009.

Bernhard Bauer, Jörg P. Muller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. In Paolo Ciancarini and Michael Wooldridge, editors, *Agent-Oriented Software Engineering*, pages 109–120. Springer Verlag Berlin Heidelberg, 2001.

Joachim Baumann. *Mobile Agents: Control Algorithms*, volume 1658 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

Joachim Baumann, Fritz Hohl, Kurt Rothermel, and Markus Straßer. Mole – Concepts of a Mobile Agent System. *World Wide Web*, 1(3):123–137, 1998.

Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE: a FIPA2000 compliant agent development environment. *AGENTS '01 Proceedings of the fifth international conference on Autonomous agents*, pages 216–217, 2001.

Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd, February 2007.

Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. JADE: A software framework for developing multi-agent applications. Lessons learned. *Information and Software Technology*, 50(1-2):10–21, January 2008.

Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli. *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. (Multiagent Systems, Artificial Societies, and Simulated Organizations). Springer US, 2004.

Carole Bernon, Massimo Cossentino, Marie-Pierre Gleizes, Paola Turci, and Franco Zambonelli. A study of some multi-agent meta-models. *Agent-Oriented Software Engineering V*, 3382:62—77, 2005a.

Carole Bernon, Massimo Cossentino, and Juan Pavón. Agent-oriented software engineering. *The Knowledge Engineering Review*, 20(2):99–116, June 2005b.

Andrzej Bieszczad, Bernard Pagurek, and Tony White. Mobile agents for network management. *IEEE Communications Surveys & Tutorials*, 1(1):2–9, January 1998.

Kiran Bondalapati, Pedro Diniz, Phillip Duncan, John Granacki, Mary Hall, Rajeev Jain, and Heidi Ziegler. DEFACTO: A design environment for adaptive computing technology. *Parallel And Distributed Processing - Lecture Notes in Computer Science*, 1586:570–578, 1999.

Rafael H. Bordini and Jomi Fred Hübner. Jason - A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net. 2004.

Rafael H. Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking AgentSpeak. *Proceedings of the second international joint conference on Autonomous agents and multiagent systems (AAMAS '03)*, 2003.

Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J. Gomez-Sanz, Joao Leite, Gregory M.P. O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *INFORMATICA*, 30(1):33–44, January 2006.

Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming.* Languages, Tools and Applications. Springer-Verlag New York Inc, June 2009.

Giampio Bracchi and Barbara Pernici. The design requirements of office systems. *ACM Transactions on Office Information Systems (TOIS)*, 2(2):151–170, 1984.

Jeffrey M. Bradshaw. An Introduction to Software Agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 3–46. MIT Press, April 1997.

Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the Association for Computing Machinery (JACM)*, 30(2):323–342, April 1983.

Michael E. Bratman. *Intention, Plans, and Practical Reason.* CSLP Publications - Center for the Study of Language and Information, 1999.

Peter Braun and Steffen Kern. Towards Adaptive Migration Techniques for Mobile Agents. In *Fifth Workshop on Adaptive Agents and Multi-Agent Systems (AAMAS 2005)*, Paris, France, April 2005.

Peter Braun and Wilhelm Rossak. *Mobile Agents.* Basic Concepts, Mobility Models, and the Tracy Toolkit. Morgan Kaufmann Publishers, December 2004.

Peter Braun, Jan Eismann, Christian Erfurth, and Wilhelm Rossak. TRACY-a prototype of an architected middleware to support mobile agents. In *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings.*, pages 255–260, Washington, DC, USA , USA, 2001. Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany.

Peter Braun, Ingo Müller, Sven Geisenhainer, Volkmar Schau, and Wilhelm Rossak. Agent Migration as an Optional Service in an Extendable Agent Toolkit Architecture. *Mobility Aware Technologies and Applications, Lecture Notes in Computer Science*, 3284:127–136, 2004.

Peter Braun, Ingo Müller, Tino Schlegel, Steffen Kern, Volkmar Schau, and Wilhelm Rossak. Tracy: An Extensible Plugin-Oriented Software Architecture for Mobile Agent Toolkits. pages 357–381. Software Agent-Based Applications, Platforms and Development Kits, Whitestein Series in Software Agent Technology and Autonomic Computing, 2005.

Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

Brian Brewington, Robert Gray, Katsuhiro Moizumi, David Kotz, George Cybenko, and Daniela Rus. Mobile agents in distributed information retrieval. In Matthias Klusch, editor, *Intelligent Information Agents: Cooperative, Rotational and Adaptive Information Gathering on the Internet*, pages 355–395. Intelligent Information Agents, 1999.

Rodney A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE journal of robotics and automation*, 2(1):14–23, March 1986.

Davide Brugali, Giuseppe Menga, and Stefano Galarraga. Inter - Company Supply Chains Integration via Mobile Agents . In *Globalization of Manufacturing in the Digital Communications Era of the 21st Century: Innovation, Agility, and the Virtual Enterprise: Proceedings of the Tenth International IFIP WG5. 2/5.3 International Conference PROLAMAT 98, Trento, Italy, September 9-*, pages 43–54, Trento, Italy, September 1998.

Erik M Buck and Donald A Yacktman. *Cocoa Design Patterns*. Addison-Wesley Professional, 1 edition, September 2009.

Paolo Busetta and Kotagiri Ramamohanarao. An architecture for mobile BDI agents. *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 445–452, 1998.

Wei Cai, Min Chen, Lei Shu, and Takahiro Hara. GA-MIP: Genetic algorithm based multiple Mobile Agents itinerary planning in wireless sensor networks. *Wireless Internet Conference (WICON), 2010 The 5th Annual ICST*, pages 1–8, March 2010.

G Caire, M Porta, E Quarantotto, and G Sacchi. Wolf–An Eclipse Plug-In for WADE. *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2008. WETICE'08. IEEE 17th*, pages 26–32, 2008a.

Giovanni Caire, Danilo Gotta, and Massimo Banzi. WADE: a software platform to develop mission critical applications exploiting agents and workflows. *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, pages 29–36, 2008b.

Monique Calisti, Petra Funk, Sven Biellman, and Thomas Bugnon. A Multi-Agent System for Organ Transplant Management. In Antonio Moreno and John L. Nealon, editors, *Applications of Software Agent Technology in the Health Care Domain*, pages 199–215. Birkhäuser Verlag, Basel-Boston-Berlin, Basel, Switzerland, 2003.

Jiannong Cao, Xinyu Feng, Jian Lu, H.C.B. Chan, and S.K. Das. Reliable message delivery for mobile agents: push or pull? *IEEE Transactions on Systems Man and Cybernetics Part A: Systems and Humans*, 34(5):577–577587, September 2004.

Junwei Cao, Darren J. Kerbyson, and Graham R. Nudd. High Performance Service Discovery in Large-scale Multi-Agent and Mobile-Agent Systems. *In International Journal of Software Engineering and Knowledge Engineering, Special Issue on Multi-Agent Systems and Mobile Agents, World Scientific*, 11(5):621–641, 2001.

Sean Carlin and Kevin Curran. Cloud Computing Security. *International Journal of Ambient Computing and Intelligence (IJACI)*, 3(1):14–19, 2011.

Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing Distributed Applications with a Mobile Code Paradigm. In *Proceedings of the 19th international* *conference on Software engineering*, pages 22–32, Boston, MA, USA, 1997. Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy.

Santhana Chaimontree, Katie Atkinson, and Frans Coenen. A Multi-Agent Based Approach To Clustering: Harnessing The Power of Agents. 2011.

Yue-Shan Chang and Chih-Tien Fan. A Mobile Agent-Based Software Intelligence Framework in Ubiquitous Environment. In *7th International Conference on Ubiquitous Intelligence & Computing and 7th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2010*, pages 76–81, Xian, Shaanxi, October 2010. Dept. of Comp. Sci. & Inf. Eng., Nat. Taipei Univ., Sanhsia, Taiwan.

Yue-Shan Chang, Pei-Chun Shih, and Yu-Cheng Luo. Adaptive Knowledge Retrieving on Mobile Grid. In *2008 Eighth International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 158–163. Dept. of Comput. Sci. & Inf. Eng., Nat. Taipei U., Taipei, IEEE, November 2008.

Yue-Shan Chang, Chao-Tung Yang, and Yu-Cheng Luo. An Ontology based Agent Generation for Information Retrieval on Cloud Environment. *Journal of Universal Computer Science*, 17(8):1135–11351160, April 2011.

Bo Chen and Harry H. Cheng. A Review of the Applications of Agent Technology in Traffic and Transportation Systems. *IEEE Transactions on Intelligent Transportation Systems*, 11(2):485–497, June 2010.

Gaoyun Chen, Jun Lu, Jian Huang, and Zexu Wu. SaaAS-The mobile agent based service for cloud computing in internet environment. *Sixth International Conference on Natural Computation (ICNC), 2010*, pages 2935–2939, September 2010.

Min Chen, Taekyoung Kwon, Yong Yuan, and Victor C.M. Leung. Mobile agent based wireless sensor networks. *Journal of Computers*, 1(1):14–21, April 2006.

Min Chen, Sergio Gonzalez, and Victor C.M. Leung. Applications and design issues for mobile agents in wireless sensor networks. *Wireless Communications*, 14(6):20–26, December 2007a.

Min Chen, Taekyoung Kwon, Yong Yuan, Yanghee Choi, and Victor C.M. Leung. Mobile agent-based directed diffusion in wireless sensor networks. *EURASIP Journal on Applied Signal Processing*, 2007(1):13, 2007b.

Wei Chen and Yi Zhang. A Multi-constrained Routing Algorithm Based on Mobile Agent for MANET Networks. *International Joint Conference on Artificial Intelligence*, pages 16–19, 2009.

David M. Chess. Security issues in mobile code systems. *Lecture Notes in Computer Science: Mobile Agents and Security*, 1419:1–14, 1998.

David M. Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, 2(5):34–49, October 1995.

David M. Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? *Lecture Notes in Computer Science*, 1222:25–45, 1997.

S. Chetan, Gautam Kumar, K. Dinesh, K. Mathew, and M.A. Abhimanyu. Cloud Computing for Mobile World.

Sung-Jin Choi, Maeng-Soon Baik, Hong-Soo Kim, Eun-Joung Byun, and Hyunseung Choo. A Reliable Communication Protocol for Multiregion Mobile Agent Environments. *IEEE Transactions on Parallel And Distributed Systems*, 21(1):72–85, January 2010.

Dimitris N. Chorafas. *Cloud Computing Strategies.* CRC Press, Inc., Boca Raton, FL, USA, 2010.

Xiaowen Chu, Xiaowei Chen, Kaiyong Zhao, and Jiangchun Liu. Reputation and trust management in heterogeneous peer-to-peer networks. *Telecommunication Systems: Security for Multimedia and Ubiquitous Applications*, 44(3-4):191–203, 2010.

David Clark. The design philosophy of the DARPA Internet protocols. *SIGCOMM '88 Symposium proceedings on Communications architectures and protocols*, pages 106–114, 1988.

Rem W. Collier, Colm F.B. Rooney, Ruadhan P.S. O'Donoghue, and Gregory M.P. O'Hare. Mobile BDI Agents. *11 th Irish Conference on Artificial Intelligence & Cognitive Science*, 2000.

George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems.* Concepts and Design. Addison Wesley, 4 edition, June 2005.

Stefan Covaci, Tianning Zhang, and Ingo Busse. Mobile intelligent agents for the management of the informationinfrastructure. *Thirty-First Annual Hawaii International Conference on System Sciences*, 7, 1998.

Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *MobiSys '10 Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, New York, NY, USA, 2010. Microsoft Research, Redmond, WA, USA.

Orhan Dagdeviren, Ilker Korkmaz, Fatih Tekbacak, and Kayhan Erciyes. A Survey of Agent Technologies for Wireless Sensor Networks. *IETE Technical Review*, 28(2): 168–184, 2011.

Jonathan Dale. *A Mobile Agent Architecture to Support Distributed Resource Information Management.* PhD thesis, University of Southampton, June 1998.

Amir Vahid Dastjerdi, Kamalrulnizam Abu Bakar, and Sayed Gholam Hassan Tabatabaei. Distributed Intrusion Detection in Clouds Using Mobile Agents. *2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 175–180, October 2009.

Janilma A.R. de V Peres and Ulf Bergmann. Experiencing AUML for MAS Modeling: A Critical View. *Proc. of Software Engineering for Agent-Oriented Systems*, pages 11–20, 2005.

Dwight Deugo. Mobile Agent Messaging Models. In *5th International Symposium on Autonomous Decentralized Systems, 2001. Proceedings.*, pages 278–286, Dallas, Texas, 2001. Carleton University.

Gianni Di Caro and Marco Dorigo. Mobile agents for adaptive routing. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, pages 74–83, Kohala Coast, HI , USA, January 1998. IRIDIA, Univ. Libre de Bruxelles, IEEE Comput. Soc.

Niels Drost, Rob V. Van Nieuwpoort, Jason Maassen, Frank J. Seinstra, and Henri E. Bai. Zorilla: a peertopeer middleware for realworld distributed systems. *Concurrency and Computation: Practice and Experience*, 23(13):1506–1521, September 2011.

Jaliya Ekanayake and Geoffrey Fox. High performance parallel computing with clouds and cloud technologies. In Dimiter R. Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, Eliezer Dekel, Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, and Geoffrey Coulson, editors, *Cloud Computing, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 20–38. Springer Berlin Heidelberg, 2010.

Christian Erfurth. *Proaktive autonome Navigation für mobile Agenten*. PhD thesis, Friedrich-Schiller-Universität Jena, Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik, November 2004.

Christian Erfurth, Steffen Kern, Wilhelm Rossak, Peter Braun, and Antje Leßmann. MobiSoft: Networked Personal Assistants for Mobile Users in Everyday Life. *Cooperative Information Agents XII, Lecture Notes in Computer Science*, 5180:147–161, 2008.

Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, Upper Saddle River, NJ, USA, 2005.

Maria Fasli. *Agent Technology For E-Commerce*. John Wiley & Sons, 1 edition, January 2007.

Yuhong Feng and Wentong Cai. Execution coordination in mobile agent-based distributed job workflow execution. *Journal of Systems Architecture*, 54(10):944–956, October 2008.

Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

Ian Foster, Nicholas R. Jennings, and Carl Kesselman. Brain meets brawn: Why grid and agents need each other. *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*, pages 8–15, 2004.

Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop*, pages 1–10, Austin, TX, December 2008. Dept. of Comput. Sci., Univ. of Chicago, Chicago, IL, USA.

Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

Munehiro Fukuda, Koichi Kashiwagi, and Shinya Kobayashi. AgentTeamwork: Coordinating grid-computing jobs with mobile agents. *Applied Intelligence: Special Issue: Agent-based Grid Computing*, 25(2):181–198, 2006.

Julia Rose Galliers. *A theoretical framework for computer models of cooperative dialogue, acknowledging multiagent conflict.* PhD thesis, Open Univ., Milton Keynes (United Kingdom), 1988.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, November 1994.

Jesse James Garrett. Ajax: A new approach to web applications. *robertspahr.com*, 2005.

Alessandro Genco, editor. *Mobile agents: principles of operation and applications.* Advances in Management Information. WIT Press Publishing, University of Palermo, Italy, 2008.

Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 147, July 1994.

Carlo Ghezzi and Giovanni Vigna. Mobile code paradigms and technologies: A case study. *Lecture Notes in Computer Science*, 1219:39–49, 1997.

Sergio González-Valenzuela, Min Chen, Huasong Cao, and Victor C.M. Leung. Programmable re-tasking of wireless sensor networks using WISEMAN. In Jun Zheng, Shiwen Mao, Scott F. Midkiff, Hua Zhu, Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, and Geoffrey Coulson, editors, *Ad Hoc Networks, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 780–794. Springer Berlin Heidelberg, 2010a.

Sergio González-Valenzuela, Min Chen, and Victor C.M. Leung. Programmable Middleware for Wireless Sensor Networks Applications Using Mobile Agents. *Mobile Networks and Applications*, 15(6):853–865, 2010b.

Sergio González-Valenzuela, Min Chen, and Victor C.M. Leung. Applications of Mobile Agents in Wireless Networks and Mobile Computing. *Advances in Computers*, 82: 113–163, 2011.

Richard Goodwin. Formalizing Properties of Agents. *Journal of Logic and Computation*, 5(6):763–781, June 1993.

Robert Gray, David Kotz, George Cybenko, and Daniela Rus. Mobile agents: Motivations and state-of-the-art systems. 2000.

Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Agent Tcl: A flexible and secure mobile-agent system. *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, pages 2–2, May 1996a.

Robert S. Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical report, Thayer School of Engineering / Department of Computer Science; Dartmouth College, 1996b.

Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software: Practice and Experience, Special Issue: Mobile Agent Systems*, 32(6):543–573, April 2002.

Robert L. Grossman. The case for cloud computing. *it Professional*, 11(2):23–27, April 2009.

J. Octavio Gutierrez-Garcia and Kwang-Mong Sim. Agent-Based Service Composition in Cloud Computing. *Grid and Distributed Computing, Control and Automation, Communications in Computer and Information Science*, 121:1–10, 2010.

Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9, July 2008.

Minghua He and Ho-fung Leung. Agents in E-Commerce: State of the Art. *Knowledge and Information Systems*, 4(3):257–282, 2002.

Abdel-Fattah Hegazy, Amr Badr, and Mohammed Kassab. Agent based cloud storage system. *Proceedings of the 10th The World Scientific and Engineering Academy and Society Conference*, pages 240–245, 2010.

Brian Henderson-Sellers and Paolo Giorgini. *Agent-Oriented Methodologies.* IGI Global, 2005, April 2005.

Klaus Herrmann. MESH Mdl—A Middleware for Self-Organization in Ad Hoc Networks. In *23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, page 446, Providence, Rhode Island, USA, May 2003. Berlin University of Technology.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. *IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, 1973.

Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4): 357–401, 1999.

Jiří Hodík, Petr Bečvář, Michael Pěchouček, Jiří Vokřínek, and Jiří Pospíšil. ExPlanTech and ExtraPlanT: multi-agent technology for production planning, simulation and extra-enterprise collaboration. *International Journal of Computer Systems Science & Engineering*, 5(20):357–367, 2005.

Fritz Hohl. *Sicherheit in Mobile-Agenten-Systemen*. PhD thesis, Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, August 2001.

Jinbiao Hou. Research on Distributed Data Mining in E-commerce Environment Based on Web Services and Mobile Agent. *2009 International Conference on Computational Intelligence and Natural Computing*, pages 259–261, 2009.

Marcus J. Huber. JAM: a BDI-theoretic mobile agent architecture. In *AGENTS '99 Proceedings of the third annual conference on Autonomous Agents*, pages 236–243, Seattle, Washington, United States, 1999. Intelligent Reasoning Systems, Oceanside, CA.

Michael N. Huhns and Munindar P. Singh. Multiagent Treatment of Agenthood. *Applied Artificial Intelligence*, 13(1-2):3–10, 1999.

Matthew M. Huntbach and Graem A. Ringwood. *Agent-oriented programming.* from prolog to guarded definite clauses. Springer-Verlag New York Inc, Department of Computer Science, Queen Mary and Westfield College, London, UK, October 1999.

Carlos A. Iglesias, Mercedes Garrijo, and José C. González. A survey of agent-oriented methodologies. *INTELLIGENT AGENTS V: AGENTS THEORIES, ARCHITEC-*

TURES, AND LANGUAGES, Lecture Notes in Computer Science, 1555:317–330, 1999.

Mogos Radu Ioan and Socoll Paula Liliana. Using Mobile Agents and Intelligent Data Analysis Techniques for Climate Environment Modeling and Weather Analysis and Prediction. In *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2008. SYNASC '08*, pages 316–319, Timisoara, September 2008. Bucharest Univ. of Econ., Bucharest, Romania.

Nicholas R. Jennings and Michael J. Wooldridge. Applications of intelligent agents. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, pages 3–28. Springer-Verlag: Heidelberg, Germany, 1998.

Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System. Technical report, June 1995.

Dag Johansen, Kåre J. Lauvset, Robbert van Renesse, Fred B. Schneider, Nils P. Sudmann, and Kjetil Jacobsen. A TACOMA retrospective. *Software: Practice and Experience, Special Issue: Mobile Agent Systems*, 32(6):605–619, 2002.

Catholijn Jonker and Valentin Robu. Automated Multi-Attribute Negotiation with Efficient Use of Incomplete Preference Information. *AAMAS '04 Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, 3:1054–1061, 2004.

Leslie Pack Kaelbling. An architecture for intelligent reactive systems. *Reasoning about Actions & Plans: Proceedings of the 1986 Workshop*, pages 395–410, February 1987.

Elizabeth A. Kendall. Role modelling for agent system analysis, design, and implementation. *First International Symposium on Agent Systems and Applications, 1999 and Third International Symposium on Mobile Agents. Proceedings*, pages 204–218, 1999.

Elizabeth A. Kendall, P.V. Murali Krishna, Chirag V. Pathak, and C.B. Suresh. Patterns of intelligent and mobile agents. *Proceedings of the second international conference on Autonomous agents*, pages 92–99, 1998.

Steffen Kern and Peter Braun. Towards Adaptive Migration Strategies for Mobile Agents. *Innovative Concepts for Autonomic and Agent-based Systems, Lecture Notes in Computer Science*, 3825:334–345, 2006.

Steffen Kern, Peter Braun, Chris Fensch, and Wilhelm Rossak. Class Splitting as a Method to Reduce Migration Overhead of Mobile Agents. *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004, Agia Napa (Cyprus), October 2004, Proceedings, Part II*, 3291:1358–1375, 2004.

Steffen Kern, Torsten Dettborn, Ronny Eckhaus, Yang Ji, Christian Erfurth, Wilhelm Rossak, and Peter Braun. Assistant-Based Mobile Supply Chain Management. *13th Annual IEEE International Symposium and Workshops on the Engineering of Computer Based Systems - Mastering the Complexity of Computer-based Systems (ECBS 2006)*, pages 23–31, March 2006a.

Steffen Kern, Torsten Dettborn, Ronny Eckhaus, Yang Ji, Christian Erfurth, Wilhelm Rossak, and Peter Braun. A Generic Agent-based Peer-to-Peer Infrastructure for Social-mobile Applications. *Mobile Informationssysteme - Potentiale, Hinternisse, Einsatz, 1. Fachtagung Mobilität und Mobile Informationssysteme (MMS 2006)*, P-76: 127–138, February 2006b.

In-Cheol Kim. An Agent-Oriented Approach to Semantic Web Services. *Next Generation Information Technologies And Systems, Lecture Notes in Computer Science*, 4032:341–344, 2006.

Myoung Kim, Hyo Yoon, and Han Lee. An Intelligent Multi-Agent Model for Resource Virtualization: Supporting Social Media Service in Cloud Computing. *Computers, Networks, Systems, and Industrial Engineering 2011, Studies in Computational Intelligence*, 365:99–111, 2011a.

Myoungjin Kim, Hanku Lee, and Wonhong Nam. A Model of Multi-agent Design for Virtualization Resource Configuration in Cloud Computing. *First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering (CNSI), 2011*, pages 234–239, May 2011b.

David Kinny, Michael Georgeff, and Anand Rao. A methodology and modelling technique for systems of BDI agents. *Agents Breaking Away, Lecture Notes in Computer Science*, 1038:56–71, 1996.

Andreas Klein, Christian Mannweiler, Joerg Schneider, and Hans D. Schotten. Access schemes for mobile cloud computing. *2010 Eleventh International Conference on Mobile Data Management*, pages 387–392, 2010.

Christian Kloch, Ebbe B. Petersen, and Ole Brun Madsen. Cloud Based Infrastructure, the New Business Possibilities and Barriers. *Wireless Personal Communications, Special Issue: Distributed and Secure Cloud Clustering (DISC) - (Selected Topics from the Strategic Workshop, May 26-28, 2010, Florence, Italy)*, 58(1):17–30, 2011.

Mamadou Tadiou Kone, Akira Shimazu, and Tatsuo Nakajima. The state of the art in agent communication languages. *Knowledge and Information Systems*, 2(3):259–284, 2000.

David Kotz and Friedemann Mattern, editors. *Agent Systems, Mobile Agents, and Applications.* Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (Lecture Notes in Computer Science). Springer, 1 edition, October 2000.

Tobias Kowatsch, Wolfgang Maass, Andreas Filler, and Sabine Janzen. Knowledge-based bundling of smart products on a mobile recomendation agent. *7th International Conference on Mobile Business, 2008. ICMB '08*, pages 181–190, July 2008.

Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA.* Sservice-Oriented Architecture Best Practices. Prentice Hall Professional, 2005.

Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing transfer delay using Java class file splitting and prefetching. *Proceings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 276–291, November 1999.

U.P. Kulkarni, P.D. Desai, J. Ahmed, J.V. Vadavi, and A. Yardi. Mobile Agent Based Distributed Data Mining. In *International Conference on Computational Intelligence and Multimedia Applications*, pages 18–24, Sivakasi, Tamil Nadu, December 2007. SDM Coll. of Eng. & Technol., Dharwad.

Yannis Labrou, Tim Finin, and Yun Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems and Their Applications*, 14(2):45–52, April 1999.

Danny B. Lange. Mobile objects and mobile agents: the future of distributed computing? *ECOOP'98 — Object-Oriented Programming, Lecture Notes in Computer Science*, 1445:1–12, 1998.

Danny B. Lange, Mitsuru Oshima, Günther Karjoth, and Kazuya Kosaka. Aglets: Programming mobile agents in Java. *Worldwide Computing and Its Applications, Lecture Notes in Computer Science*, 1274:253–266, 1997.

G Lawton. New ways to build rich internet applications. *Computer*, 41(8), 2008.

Stevens Le Blond, Fabrice Le Fessant, and Erwan Le Merrer. Finding Good Partners in Availability-Aware P2P networks. *Stabilization, Safety, and Security of Distributed Systems - Lecture Notes in Computer Science*, 5873:472–484, 2009.

Thomas Ledoux and Noury M.N. Bouraqadi-Saadani. Adaptability in Mobile Agent Systems using Reflection. *Middleware'2000 Workshop on Reflective Middleware (RM2000)*, 2000.

Habin Lee, Patrik Mihailescu, and John Shepherdson. Realising Team-Working in the Field: An Agent-based Approach. *IEEE Pervasive Computing*, 6(2):85–92, April 2007.

Jill Fain Lehman, John E. Laird, and Paul Rosenbloom. A Gentle Introduction to Soar: An Architecture For Human Cognition. In Daniel N. Osherson, Don Scarborough, and Lila R. Gleitman, editors, *An Invitation to Cognitive Science: Methods, models, and conceptual issues*, pages 211–253. 1998.

Jun Lei, Xiaoming Fu, and Diter Hogrefe. D-MORE: Dynamic mesh-based overlay peer-to-peer infrastructure. *Computer Communications*, 33(10):1191–1201, June 2010a.

Jun Lei, Lei Shi, and Xiaoming Fu. An experimental analysis of Joost peer-to-peer VoD service. *Peer-to-Peer Networking and Applications*, 3(4):351–362, 2010b.

Frank Leymann and Dieter Roller. *Production Workflow.* Concepts and Techniques. Prentice Hall, 2000.

Xuhui Li, Hao Zhang, and Yongfa Zhang. Deploying Mobile Computation in Cloud Service. *Cloud Computing, Lecture Notes in Computer Science*, 5931:301–311, 2009.

Ai-zhen Liu, Jia-zhen Wang, and Xi-hong Zhang. Mobile Agent Multi-task Scheduling Algorithm. *Computer Engineering, CNKI Journal*, 2008.

Zhenyu Liu, Tiejiang Liu, Tun Lu, Lizhi Cai, and Genxing Yang. Agent-Based Online Quality Measurement Approach in Cloud Computing Environment. *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pages 686–690, 2010.

Ignacio Lopez-Rodriguez and Mario Hernandes-Tejera. Software Agents as Cloud Computing Services. *Advances on Practical Applications of Agents and Multiagent Systems, Advances in Intelligent and Soft Computing*, 88:271–276, 2011.

Gehao Lu, Joan Lu, and Shaowen Yao. Mobile agent based data fusion for wireless sensor networks with a XML framework. *The Open Information Systems Journal*, 3: 136–145, 2009.

Michael Luck and Mark d'Inverno. A formal framework for agency and autonomy. *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260, 1995.

Michael Luck and Mark d'Inverno. A conceptual framework for agent definition and development. *The Computer Journal*, 44(1):1–20, 2001.

Pattie Maes. *Designing autonomous agents.* theory and practice from biology to engineering and back. The MIT Press, 1990.

Thomas Magedanz, Christoph Bäumer, Sheung-On Choy, and Markus Breugst. Grasshopper - a universal agent platform based on OMG MASIF and FIPA standards. Technical report, IKV++ GmbH, Kurfürstendamm 173-174, d-10707 Berlin, Germany, Berlin, 1999.

Sunil S. Manvi and Pallapa Venkataram. Applications of agent technology in communications: a review. *Computer Communications*, 27(15):1493–1508, 2004.

Sunil S. Manvi and Pallapa Venkataram. Mobile agent based approach for QoS routing. *IET Communications*, 1(3):430–439, June 2007.

Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. A case study-based comparison of web testing techniques applied to AJAX web applications. *International Journal on Software Tools for Technology Transfer*, 10(6):477–492, 2008.

Felix Gómez Mármol and Gregorio Martínez Pérez. Towards pre-standardization of trust and reputation models for distributed and heterogeneous systems. *Computer Standards & Interfaces*, 32(4):185–196, June 2010.

Daniel Massaguer, Chien-Liang Fok, Nalini Venkatasubramanian, Gruia-Catalin Roman, and Chenyang Lu. Exploring sensor networks using mobile agents. *AAMAS '06 Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 323–325, 2006.

Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating Highly Modular Java Systems.* Addison-Wesley Professional, 1 edition, February 2010.

Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE INTELLIGENT SYSTEMS*, 16(2):46–53, April 2001.

Estanisiao Mercadal, Sergi Robles, Ramon Martí, Cormac J. Sreenan, and Joan Borrell. Heterogeneous Multiagent Architecture for Dynamic Triage of Victims in Emergency Scenarios. *Advances on Practical Applications of Agents and Multiagent Systems, Advances in Intelligent and Soft Computing*, 88:237–246, 2011.

Samir Moalla, Sana Hamdi, and Bruno Defude. A New Trust Management Model in P2P Systems. *2010 Sixth International Conference on Signal-Image Technology and Internet Based Systems*, pages 241–246, December 2010.

Chayapol Moemeng, Vladimir Gorodetsky, Ziye Zuo, Yong Yang, and Chengqi Zhang. Agent-based distributed data mining: A Survey. In *Data Mining and Multi-agent Integration*, pages 47–58. Springer US, Boston, MA, 2009.

Scott Oaks. *Java Security.* O'Reilly Media, 2001.

Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam Jahanian. Virtualized in-cloud security services for mobile devices. *MobiVirt '08 Proceedings of the First Workshop on Virtualization in Mobile Computing*, 2008.

James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending UML for agents. *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence (AOIS Worshop at AAAI)*, pages 3–17, 2000.

James O'Shea, Ngoc Thanh Nguyen, Keeley Crockett, Robert J. Howlett, and Lakhmi C. Jain. *Agent and Multi-Agent Systems: Technologies and Applications*, volume 6682 of *5th KES International Conference, KES-AMSTA 2011, Manchester, UK, June 29–July 1, 2011, Proceedings.* Springer Verlag, 1st edition, 2011.

K. Ota, M. Dong, J. Wang, and S. Guo. Dynamic Itinerary Planning for Mobile Agents with a Content-Specific Approach in Wireless Sensor Networks. *IEEE 72nd Vehicular Technology Conference Fall (VTC 2010-Fall)*, pages 1–5, September 2010.

John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley Longman, Amsterdam, May 1994.

Abdelkader Outtagarts. Mobile Agent-based Applications: a Survey. *International Journal of Computer Science and Network Security*, 9(11):331–339, November 2009.

Lin Padgham and Michael Winikoff. *Developing intelligent agent systems.* A Practical Guide. John Wiley & Sons, June 2004.

Heman Pathak, Nipur, and Kumkum Garg. A Fault Tolerant Comparison Internet Shopping System: BestDeal by Using Mobile Agent. *2009 International Conference on Information Management and Engineering*, pages 541–545, 2009.

Nita Patil, Chhaya Das, Shreya Patankar, and Kshitija Pol. Analysis of Distributed Intrusion Detection Systems Using Mobile Agents. *First International Conference on Emerging Trends in Engineering and Technology, 2008. ICETET '08*, pages 1255–1260, July 2008.

Hervé Paulino and Luís Lopes. A service-oriented language for programming mobile agents . In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems - AAMAS '06*, pages 1294–1296, New York, New York, USA, 2006. ACM Press.

Hervé Paulino, Luís Lopes, and Fernando Silva. Mob: a scripting language for programming web agents. *13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany*, pages 50–61, 2003.

L D Paulson. Building rich web applications with Ajax. *Computer*, 38(10):14–17, 2005.

Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. *Lecture Notes in Computer Science*, 1219:50–61, 1997.

Gleb Peregud, Julian Zubek, Marcin Paprzycki, and Maria Ganzha. Agent-based monitoring system for cloud/Grid computing. pages 64–65. Workshop on Applications of Software Agents, 2011.

Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 1962.

David Picard and Matthieu Cord. Performances of Mobile-Agents for Interactive Image Retrieval. *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 581–586, 2006.

Alexander Pokahr and Lars Braubach. From a Research to an Industry-Strength Agent Platform: JADEX V2. 2009.

Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: Multiagent Systems, Artificial Societies and Simulated Organizations*, pages 149–174. Springer Verlag, 2005.

Alexander Pokahr, Lars Braubach, and Kai Jander. Unifying Agent and Component Concepts. *Multiagent System Technologies*, pages 100–112, 2010.

Talal Rahwan, Tarek Rahwan, Iyad Rahwan, and Ronald Ashri. Towards a mobile intelligent assistant: Agentspeak(L) agents on mobile devices. *Proceedings of the 5th International Bi-Conference Workshop on Agent Oriented Information Systems*, pages 9–15, 2003.

Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 91–104, 1997.

Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. *Lecture Notes in Computer Science*, 1038:42–55, 1996.

Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, 1991.

M.H. Raza and M.A. Shibli. Mobile agent middleware for multimedia services. *The 9th International Conference on Advanced Communication Technology*, pages 1109–1114, February 2007.

Sara Rodríguez, Dante I. Tapia, Eladio Sanz, Carolina Zato, Fernando de la Prieta, and Oscar Gil. Cloud Computing Integrated into Service-Oriented Multi-Agent Architecture. *Balanced Automation Systems for Future Manufacturing Networks, IFIP Advances in Information and Communication Technology*, 322:251–259, 2010.

Manuel Rodriguez-Perez, Oscar Esparza, and Jose L. Muñoz. Surework: a super-peer reputation framework for p2p networks. *SAC '08 Proceedings of the 2008 ACM symposium on Applied computing*, pages 2019–2023, 2008.

Colm Rooney, Rem W. Collier, and Gregory M.P. O'Hare. VIPER: A visual protocol editor. *Coordination Models and Languages, Lecture Notes in Computer Science*, 2949: 279–293, 2004.

Volker Roth. Concepts and architecture of a security-centric mobile agent server. *Autonomous Decentralized Systems*, 2001.

Volker Roth. Obstacles to the adoption of mobile agents. *Proc. of the IEEE International Conference on Mobile Data Management (MDM'04)*, pages 296–297, 2004.

Stuart Jonathan Russell and Peter Norvig. *Artificial Intelligence.* A Modern Approach. Prentice Hall, 2010.

M.D. Sadek, P. Bretier, and F. Panaget. ARTIMIS: Natural dialogue meets rational agency. *IJCAI'97 Proceedings of the Fifteenth international joint conference on Artifical intelligence - Volume 2*, pages 1030–1035, 1997.

Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. *Mobile Agents and Security, Lecture Notes in Computer Science*, 1419:44–60, 1998.

Tino Schlegel, Peter Braun, and Ryszard Kowalczyk. Towards autonomous mobile agents with emergent migration behaviour. *AAMAS '06 Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 585–592, 2006.

N. Schurr, J. Marecki, J.P. Lewis, M. Tambe, and P. Scerri. The defacto system: Coordinating human-agent teams for the future of disaster response. *Multi-Agent Programming, Multiagent Systems, Artificial Societies and Simulated Organizations*, 15: 197–215, 2005.

Zhidong Shen and Qiang Tong. A security technology for mobile agent system improved by trusted computing platform. *Ninth International Conference on Hybrid Intelligent Systems, 2009. HIS '09*, pages 46–50, August 2009.

Zhong Shen, Renfa Li, and Juan Luo. Mobile Agent Based Middleware Using Publish/Subscribe Mechanism in Wireless Sensor Networks. *2009 International Conference on Communication Software and Networks*, pages 111–115, 2009.

Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

Liang Shuang. The Design and Realization of Cloud Computing Framework Model Based on SOA. *Advanced Materials Research, Engineering Materials, Energy, Management and Control*, 171-172:696–701, December 2010.

Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, C-29(12):1104–1113, December 1980.

Katarina Stanoevska-Slabeva, Thomas Wozniak, and Santi Ristol, editors. *Grid and Cloud Computing.* A Business Perspective on Technology and Applications. Springer Verlag, 2010.

Robert Steele, Tharam Dillon, Parth Pandya, and Yuri Ventsov. XML-Based Mobile Agents. *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, pages 42–48, April 2005.

V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogeneous Agent Systems.* The MIT Press, 1 edition, June 2000.

Nils P. Sudmann and Dag Johansen. Adding mobility to non-mobile web robots. *Proceedings of the 20th International Conference on Distributed Computing Systems ICDCS'00, Workshop of Knowledge Discovery and Data Mining in the World-Wide Web*, pages F73–F79, 2000.

Nils Peter Sudmann. *TACOMA–fundamental abstractions supporting agent computing in a distributed environment.* PhD thesis, Department of Computer Science, University of Tromso, Norway, August 1996.

Ron Sun, editor. *Cognition and Multi-Agent Interaction.* From Cognitive Modeling to Social Simulation. Cambridge University Press, March 2006.

Alexandru Suna, Gilles Klein, and Amal El Fallah Seghrouchni. Using mobile agents for resource sharing. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004. (IAT 2004). Proceedings*, pages 389–392. Paris Univ. 6, France, IEEE, 2004.

Leo Szumel, Jason LeBrun, and John D. Owens. Towards a mobile agent framework for sensor networks. *EmNets '05 Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pages 79–87, 2005.

Joseph Tardo and Luis Valente. Mobile Agent Security and Telescript. *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pages 58–63, February 1996.

Lang Tong, Qing Zhao, and Srihari Adireddy. Sensor networks with mobile agents. *IEEE Military Communications Conference, 2003. MILCOM 2003.*, 1:688–693, October 2003.

Ivan Trencansky and Radovan Cervenka. Agent Modeling Language (AML): A comprehensive approach to modeling MAS. *INFORMATICA*, 29(4):391–400, January 2005.

Chang-Chun Tsai, Cheng-Jung Lee, and Shung-Ming Tang. The web 2.0 movement: mashups driven and web services. *WSEAS Transactions on Computers*, 8(8):1235–1244, August 2009.

Yuh-Min Tseng and Fu-Gui Chen. A free-rider aware reputation system for peer-to-peer file-sharing networks. *EXPERT SYSTEMS WITH APPLICATIONS*, 38(3):2432–2440, March 2011.

Amund Tveit. jfipa-an Architecture for Agent-based Grid Computing. *Proceedings of AISB'02 Convention, Symposium on AI and Grid Computing*, 2002.

H. Van Dyke Parunak. Manufacturing Experience With the Contract Net. *AI magazine*, 8(2):285–310, 1987.

H. Van Dyke Parunak. A Practitioners' Review of Industrial Agent Applications . *Autonomous Agents and Multi-Agent Systems*, 3(4):389–407, 2000.

Do Van Thanh. Using Mobile Agents in Telecommunications. *Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, page 685, 2001.

Giovanni Vigna. *Mobile Code Technologies, Paradigms, and Applications.* PhD thesis, Politecnico Di Milano, 1997.

Jeffrey Voas and Jia Zhang. Cloud Computing: New Wine or Just a New Bottle? *it Professional*, 11(2):15–17, March 2009.

Dennis Volpano and Geoffrey Smith. Language issues in mobile program security. *Mobile Agents and Security, Lecture Notes in Computer Science*, 1419:25–43, 1998.

Quang Hieu Vu, Mihai Lupu, and Beng Chin Ooi. *Peer-to-Peer Computing.* Principles and Applications. Springer Verlag, 2010.

Andrzej Walczak, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Augmenting BDI Agents with Deliberative Planning Techniques. *Programming Multi-Agent Systems: Lecture Notes in Computer Science*, 4411:113–127, 2007.

H. Q. Wang, Z. Q. Wang, Q. Zhao, G. F. Wang, R. J. Zheng, and D. X. Liu. Mobile Agents for Network Intrusion Resistance. *Advanced Web and Network Technologies, and Applications, Lecture Notes in Computer Science*, 3842:965–970, 2006.

Lizhe Wang, Gregor von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. Cloud Computing: A Perspective Study. *New Generation Computing*, 28(2):137–146, 2010.

Zhiliang Wang, Xiangjie Qiao, and Yinggang Xie. An Emotional Intelligent E-learning System Based on Mobile Agent Technology. *ICCET '09 International Conference on Computer Engineering and Technology*, pages 51–54, January 2009.

Mark N. Wayne. *Flowchart Concepts and Data Processing Techniques.* A Self-instructional Guide. Wellesley, Mass., Honeywell Institute of Information Sciences, April 1973.

Gerhard Weiss, editor. *Multiagent Systems.* A Modern Approach to Distributed Artificial Intelligence. The MIT Press, March 1999.

James E. White. Telescript technology: The foundation for the electronic marketplace. White paper. Technical report, 1994.

James E. White. Mobile agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 437–472. MIT Press Cambridge, 1997.

James E. White. Telescript technology: Mobile agents. In Dejan S Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility: Processes, Computers, and Agents*, page 704. ACM Press, April 1999.

Mark F. Wood and Scott A. DeLoach. An overview of the multiagent systems engineering methodology. *Agent-oriented Software Engineering, Lecture Notes in Computer Science*, 1957:1–53, 2001.

Michael Wooldridge. Intelligent Agents. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 27–78. The MIT Press, 1999.

Michael Wooldridge. *An Introduction to MultiAgent Systems.* Wiley & Sons, 2 edition, May 2009.

Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, July 1995.

Michael Wooldridge, Nicholas R. Jennings, and David Kinny. A methodology for agent-oriented analysis and design. *AGENTS '99 Proceedings of the third annual conference on Autonomous Agents*, pages 69–76, 1999.

Guowei Wu, Heqian Li, and Lin Yao. A Group-Based Mobile Agent Routing Protocol for Multitype Wireless Sensor Networks. *IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 42–49, December 2010.

Jing Xu and Yongzhong Li. A New Distributed Intrusion Detection Model Based on Immune Mobile Agent. *Asia-Pacific Conference on Information Processing*, pages 461–464, July 2009.

Ke Xu, Meina Song, Xiaoqi Zhang, and Junde Song. A Cloud Computing Platform Based on P2P. *ITIME '09. IEEE International Symposium on IT in Medicine & Education*, pages 427–432, August 2009.

Yingyue Xu and Hairong Qi. Modeling of Agent Migration for Collaborative and Distributed Processing. *ICNSC '06. Proceedings of the 2006 IEEE International Conference on Networking, Sensing and Control*, pages 825–830, 2006.

Takafumi Yamaya, Toramatsu Shintani, Tadachika Ozono, Yusuke Hiraoka, Hiromitsu Hattori, Takayuki Ito, Naoki Fukuta, and Kyoji Umemura. MiNet: Building Ad-Hoc Peer-to-Peer Networks for Information Sharing Based on Mobile Agents. *Practical Aspects of Knowledge Management, Lecture Notes in Computer Science*, 3336:59–70, 2004.

Jian Yang, Haihang Wang, Jian Wang, Chengxiang Tan, and Dingguo Yu. Provable Data Possession of Resource-constrained Mobile Devices in Cloud Computing. *Journal of Networks*, 6(7):1033–1040, July 2011.

William J. Yeager and Rita Y. Chen. Peer trust evaluation using mobile agents in peer-to-peer networks. *US Patent 7,213,047*, 2007.

Christopher S. Yoo. The changing patterns of Internet usage. *Federal Communications Law Journal*, 63(1):24, 2010.

Ma Yubao and Ding Renyuan. Mobile Agent Technology and Its Application in Distributed Data Mining. *First International Workshop on Database Technology and Applications*, pages 151–155, April 2009.

Franco Zambonelli and Andrea Omicini. Challenges and Research Directions in Agent-Oriented Software Engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3): 253–283, 2004.

Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, 2003.

Zehua Zhang and Xuejie Zhang. Realization of open cloud computing federation based on mobile agent. *ICIS 2009. IEEE International Conference on Intelligent Computing and Intelligent Systems*, pages 642–646, 2009.

Hong Zhou, Hongji Yang, and Andrew Hugill. An Ontology-Based Approach to Reengineering Enterprise Software for Cloud Computing. *IEEE 34th Annual Computer Software and Applications Conference*, pages 383–388, July 2010a.

Yanbo Zhou, Yichao Yang, Lei Liang, Dan He, and Zhili Sun. An Agent-based Scheme for Supporting Service and Resource Management in Wireless Cloud. In *2010 9th International Conference on Grid and Cloud Computing (GCC 2010)*, pages 34–39. Centre for Commun. Syst. Res., Univ. of Surrey, Guildford, UK, IEEE, November 2010b.

Roland Zimmermann, Stefan Winkler, and Freimut Bodendorf. Supply Chain Event Management With Software Agents. In Stefan Kirn, Otthein Herzog, Peter Lockemann, and Otto Spaniol, editors, *International Handbooks on Information Systems, Multiagent Engineering*, pages 157–175. Springer Verlag, Berlin/Heidelberg, 2006.

Ehrenwörtliche Erklärung

Hiermit erkläre ich

- dass mir die Promotionsordnung der Fakultät bekannt ist,
- dass ich die Dissertation selbst angefertigt habe, keine Textabschnitte oder Ergebnisse eines Dritten oder eigene Prüfungsarbeiten ohne Kennzeichnung übernommen und alle von mir benutzten Hilfsmittel, persönliche Mitteilungen und Quellen in meiner Arbeit angegeben habe,
- dass ich die Hilfe eines Promotionsberaters nicht in Anspruch genommen habe und dass Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhalten haben, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen,
- dass ich die Dissertation noch nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht habe.

Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts haben mich folgende Personen unterstützt: Niemand.

Ich habe die gleiche, eine in wesentlichen Teilen ähnliche bzw. eine andere Abhandlung bereits bei einer anderen Hochschule als Dissertation eingereicht: Nein.

Jena, den ....

Unterschrift